- Java并发开发、互联网项目开发必备书籍
- 线程，线程安全，线程集合类，线程阀，线程池，Fork/Join，线程、线程池在互联网项目开发的应用，线程监控及线程分析，Android中线程应用

Java并发编程从入门到精通

本书示例源代码

张振华 著

清华大学出版社

Java并发编程从入门到精通

张振华 著

清华大学出版社
北京

内 容 简 介

本书作者结合自己 10 多年 Java 并发编程经验，详细介绍了 Java 并发编程的基础概念、工作原理、编程技巧和注意事项，对 Java 高性能高并发编程有极大的参考价值。

本书内容包括并发编程概念，线程，线程安全，线程集合类，线程阀，线程池，Fork/Join，线程、线程池在互联网项目开发的应用，线程监控及线程分析，Android 中线程应用。

本书适合 Java 开发初学者， Java 开发工程师，以及 Java 网络应用优化人员使用，也适合高校相关专业的师生作为课程设计参考使用。

图书在版编目（CIP）数据

Java 并发编程从入门到精通 / 张振华著. -- 北京 ：清华大学出版社，2015
ISBN 978-7-302-40191-9

I. ①J… II. ①张… III. ①JAVA 语言－程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2015）第 101591 号

责任编辑：夏非彼
封面设计：王 翔
责任校对：闫秀华
责任印制：沈 露

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦 A 座 邮 编：100084
社 总 机：010-62770175 邮 购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：清华大学印刷厂
经 销：全国新华书店
开 本：190mm×260mm **印 张**：14 **字 数**：358 千字
版 次：2015 年 7 月第 1 版 **印 次**：2015 年 7 月第1 次印刷
印 数：1～3000
定 价：39.00 元

产品编号：064079-01

序 言

古时候，有一个自认为佛学造诣很深的人，听说某个寺庙里有位德高望重的老禅师，便去拜访。老禅师十分恭敬地接待了他，他讲了自己的很多心得，希望老禅师给予指点。

老禅师听后，没有说话，只是为他沏茶。可是在倒水时，明明水已经满了，老禅师还在倒，而不顾茶水都已经溢了出来。最后，这个人终于忍不住说："大师，杯子已经满了。"老禅师这才住手。这个人问老禅师："大师，请你指点。"老禅师说："我已经教你了。"

这个人不明所以，只好回去了。冥思苦想，终于有一天他想明白了：如果自己不把旧茶倒掉，又哪有空间来添续新茶？

空杯心态不仅是一种心境，更是一种做人的境界。其实我们学习和看任何一本书的时候，如果以空杯的心态去看的话，相信收获会更多。功夫巨星李小龙就非常推崇空杯心态，他说："清空你的杯子，方能再行注满，空无以求全。"

最近发现市面上有些书籍要不就是直译过来的，很多不实用，要不就是讲的太玄乎其神了，而此书换一种讲解方式和思路来理解多并发和多线程，让多线程、多并发没有那么玄乎。作者以 10 年的开发经验做总结，希望能帮助读者少走一些弯路，读完这本书让菜鸟变大牛。本书内容安排由浅入深再到应用实践。作者建议大家，不要动不动就 Hadoop，动不动就分布式，将 Java 里面的多并发编程掌握好了，其实就可以解决很多应用问题。

建议大家看此书的时候，结合 JDK 的源码，一起看，每个实例都要运行看看，还要看看咱们工作中，相关的设计是否合理。纸上得来终觉浅，绝知此事要躬行。一定要多加练习才行。

书上有一部分内容是应网友要求编写的，在此表示感谢！也感谢为本书提供精彩书评的朋友。谢谢大家的支持！

本书示例源代码下载地址如下：

http://pan.baidu.com/s/1mgzIbX2

如果下载有问题，请电子邮件联系 booksaga@163.com，邮件主题为"求 JAVA 并发代码"。

著者

2015 年 5 月

推荐语

我在IT软件行业从业已12年。作为“前辈”，衡量一名“程序猿”的技术实力，一般会看你是否具备深度的系统性能调优的能力。云计算的时代，对系统的高性能、高并发要求更高。所以，深入了解和掌握Java的多线程机制原理，非常有用，非常必要。

这本书的所有知识均来自于作者多年的项目实践，倾注了作者多年的心血。讲解的深入浅出，让你掌握起来毫不费力。如果你想成为一名架构师，如果你想成为一名资深的技术大牛，强烈推荐你读一读，你值得拥有！

多家IT公司担任研发总监、技术总监 Justin.Han（韩剑锋）

在进行并发编程开发之前，深入学习并发理论知识非常有必要，比如阅读并发容器的源码。本书通过大量代码实例，讲解并发知识，非常细致。而在实战中并发程序的问题定位也是非常麻烦，相信本书也能给初学者一些参考。

阿里巴巴资深开发工程师，ifeve.com创始人 Kiral（方腾飞）

目 录

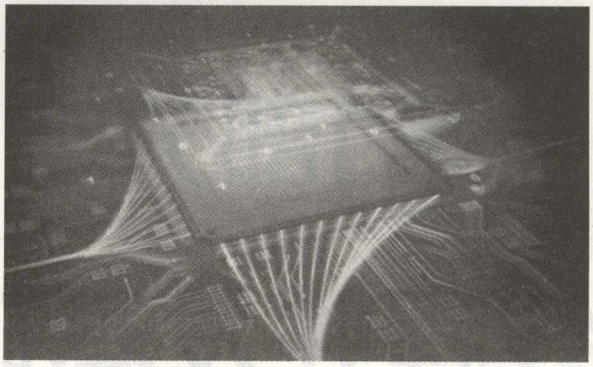
第 1 部分　线程并发基础

第1部分

线程并发基础

第 1 章
概念部分

关注细节求本质，把握机会促发展。



理解互联 Web 开发过程中并发编程相关的必须要知道的一些概念。

1.1 CPU 核心数、线程数

位宽（32 位与 64 位 CPU）32/64 位指的是 CPU 位宽，更大的 CPU 位宽有两个好处：一次能处理更大范围的数据运算和支持更大容量的内存。一般情况下 32 位 CPU 只支持 4GB 以内的内存，更大容量的内存无法在系统识别（服务器级除外）。于是就有了 64 位 CPU，然后就有了 64 位操作系统与软件。

- 多核心：也指单芯片多处理器（Chip Multiprocessors，简称 CMP）。CMP 是由美国斯坦福大学提出的，其思想是将大规模并行处理器中的 SMP（对称多处理器）集成到同一芯片内，各个处理器并行执行不同的进程。这种依靠多个 CPU 同时并行地运行程序是实现超高速计算的一个重要方向，称为并行处理。
- 多线程：Simultaneous Multithreading，简称 SMT。SMT 可通过复制处理器上的结构状态，让同一个处理器上的多个线程同步执行并共享处理器的执行资源，可最大限度地实现宽发射、乱序的超标量处理，提高处理器运算部件的利用率，缓和由于数据相关或 Cache 未命中带来的访问内存延时。当没有多个线程可用时，SMT 处理器几乎和传统的宽发射超标

量处理器一样。SMT 最具吸引力的是只需小规模改变处理器核心的设计，几乎不用增加额外的成本就可以显著地提升效能。多线程技术则可以为高速的运算核心准备更多的待处理数据，减少运算核心的闲置时间。

- 核心数、线程数：目前主流 CPU 有双核、三核和四核，六核也在 2010 年发布。增加核心数目就是为了增加线程数，因为操作系统是通过线程来执行任务的，一般情况下它们是 1:1 对应关系，也就是说四核 CPU 一般拥有四个线程。但 Intel 引入超线程技术后，使核心数与线程数形成 1:2 的关系。

下面我们来看看主流 CPU 的核心数线程数概况。

个人 PC 机，一般是只有一个 CPU：

Intel奔腾双核	Intel酷睿i3	Intel酷睿i5	Intel酷睿i7
双核心 双线程	双核心 四线程	双核心 四线程 四核心 四线程	四核心 八线程 六核心 十二线程 八核心 十六线程

服务器 CPU，与 PC 机的个人电脑的最大不一样是可以有多个 CPU：

IntelXeon 5600	IntelXeon E3	IntelXeon E5	IntelXeon E7
四核心 八线程 六核心 十二线程	双核心 四线程 四核心 八线程	双核心 四线程 四核心 四线程 六核心 十二线程 八核心 十六线程 十二核心 二十四线程	六核心 十二线程 八核心 十六线程 十核心 二十线程

Linux 和 Windows 都有相关的查询以上参数的方法：

Windows 里面查看 CPU 的物理信息，其实就相对简单了，直接在任务管理器里面和设备管理里面就可以看得出来，或者通过第三方工具“鲁大师”等等。

Linux 里面查询 CPU 的物理信息，通过/proc/cpuinfo 这个文件即可查看详情。

（1）查询 CPU 的型号：

```
[root]#  cat /proc/cpuinfo | grep name | cut -f2 -d: | uniq -c
4  Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz
```

可以得出 CPU 的型号是 E5 系列的。

（2）查看 CPU 有几个核几个线程：

```
[root]# grep 'processor' /proc/cpuinfo | sort -u | wc -l
4
[root]# grep 'core id' /proc/cpuinfo | sort -u | wc -l
2
```

可以看得出来有 4 个线程，2 个 CPU 核。

了解你的服务器硬件信息，特别是 CPU 信息，对于做高并发的开发来者来说有益无碍。

1.2 CPU 时间片轮转机制

时间片轮转调度是一种最古老、最简单、最公平且使用最广的算法，又称 RR 调度。每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。

百度百科对 CPU 时间片轮转机制原理解释如下：

如果在时间片结束时进程还在运行，则 CPU 将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则 CPU 当即进行切换。调度程序所要做的就是维护一张就绪进程列表，当进程用完它的时间片后，它被移到队列的末尾。

时间片轮转调度中唯一有趣的一点是时间片的长度。从一个进程切换到另一个进程是需要一定时间的，包括保存和装入寄存器值及内存映像，更新各种表格和队列等。假如进程切（process witch），有时称为上下文切换（context switch），需要 5ms，再假设时间片设为 20ms，则在做完 20ms 有用的工作之后，CPU 将花费 5ms 来进行进程切换。CPU 时间的 20%被浪费在了管理开销上了。

为了提高 CPU 效率，我们可以将时间片设为 5000ms。这时浪费的时间只有 0.1%。但考虑到在一个分时系统中，如果有 10 个交互用户几乎同时按下回车键，将发生什么情况？假设所有其他进程都用足它们的时间片的话，最后一个不幸的进程不得不等待 5s 才获得运行机会。多数用户无法忍受一条简短命令要 5s 才能做出响应。同样的问题在一台支持多道程序的个人计算机上也会发生。

结论可以归结如下：时间片设得太短会导致过多的进程切换，降低了 CPU 效率；而设得太长又可能引起对短的交互请求的响应变差。将时间片设为 100ms 通常是一个比较合理的折衷。

在 CPU 死机的情况下，其实大家不难发现当运行一个程序的时候把 CPU 给弄到了 100%，再不重启电脑的情况下，其实我们还是有机会把它 Kill 掉的，我想也正是因为这种机制的缘故。

1.3 什么是进程和什么是线程

1. 进程是程序运行资源分配的最小单位

进程是操作系统进行资源分配的最小单位，其中资源包括：CPU、内存空间、磁盘 IO 等，同一进程中的多条线程共享该进程中的全部系统资源，而进程和进程之间是相互独立的。进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是系统进行资源分配和调度的一个独立单位。

进程是程序在计算机上的一次执行活动。当你运行一个程序，你就启动了一个进程。显然，

程序是死的、静态的，进程是活的、动态的。进程可以分为系统进程和用户进程。凡是用于完成操作系统的各种功能的进程就是系统进程，它们就是处于运行状态下的操作系统本身，用户进程就是所有由你启动的进程。

2. 线程是 CPU 调度的最小单位，必须依赖于进程而存在

线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的、能独立运行的基本单位。线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器，一组寄存器和栈），但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

3. 线程无处不在

任何一个程序都必须要创建线程，特别是 Java 不管任何程序都必须启动一个 main 函数的主线程；Java Web 开发里面的定时任务、定时器、JSP 和 Servlet、异步消息处理机制，远程访问接口 RMI 等，任何一个监听事件，onclick 的触发事件等都离不开线程和并发的知识。

1.4 进程与线程比对

以沙箱为例进行对比阐述。一个进程就好比一个沙箱。线程就如同沙箱中的孩子们。孩子们在沙箱子中跑来跑去，并且可能将沙子攘到别的孩子眼中，他们会互相踢打或撕咬。但是，这些沙箱略有不同之处就在于每个沙箱完全由墙壁和顶棚封闭起来，无论箱中的孩子如何狠命地攘沙，他们也不会影响到其他沙箱中的其他孩子。因此，每个进程就像一个被保护起来的沙箱，未经许可，无人可以进出。

如下表所示，进程与线程的对比理解：

	进程	线程
定义	进程是程序运行的一个实体（包括：程序段、相关的数据段、进程控制块 PCB）的运行过程，是系统进行资源分配和调度的一个独立单位	线程是进程运行和执行的最小调度单位
活泼性	不活泼（只是线程的容器）	活泼，随时可以创建和销毁
系统开销	创建、撤消、切换开销大，资源要重新分配和收回	相对于进程仅保存少量寄存器内容，开销小在进程的地址空间执行代码
拥有资产	资源拥有的基本单位	相对于进程来说基本上不拥有资源，但会占用 CPU
地址空间	系统赋予的独立的内存地址空间	线程只由相关堆栈（系统栈或用户栈）寄存器和线程控制表 TCB 组成，寄存器可被用来存储线程内的局部变量
调度	仅是资源分配的基本单位	独立调度、分派的基本单位
安全性	进程之间相对比较独立，彼此不会互相影响	线程共享同一个进程下面的资源，可以相互通信和互相影响

1.5 什么是并行运行

并行运行可以简单做如下理解。

- 第一种情况：程序同时所开启的运行中的线程数，<=CPU 数量*CPU 的核心数量。
- 第二种情况：程序同时所开启的运行中的线程数，<=CPU 数量*CPU 的线程数量。

我们举个例子，如果有条高速公路 A 上面并排有 8 条车道，那么最大的并行车辆就是 8 辆，此条高速公路 A 同时并排行走的车辆小于等于 8 辆的时候，车辆就可以并行运行。CPU 也是这个原理，一个 CPU 相当于一个高速公路 A，核心数或者线程数就相当于并排可以通行的车道；而多个 CPU 就相当于并排有多条高速公路，而每个高速公路并排有多个车道。

1.6 什么是并发运行

当谈论并发的时候一定要加个单位时间，也就是说单位时间内并发量是多少？离开了单位时间其实是没有意义的。

简单来说可以做如下理解：

- 第一种情况：程序同时所开启的运行中的线程数 > CPU 数量*CPU 的核心数量。
- 第二种情况：程序同时所开启的运行中的线程数 > CPU 数量*CPU 的线程数量。

俗话说，一心不能二用，这对计算机也一样，原则上一个 CPU 只能分配给一个进程，以便运行这个进程。我们通常使用的计算机中只有一个 CPU，也就是说只有一颗心，要让它一心多用，同时运行多个进程，就必须使用并发技术。

实现并发技术相当复杂，最容易理解的是“时间片轮转进程调度算法”，它的思想简单介绍如下： 在操作系统的管理下，所有正在运行的进程轮流使用 CPU，每个进程允许占用 CPU 的时间非常短（比如 10ms），这样用户根本感觉不出来 CPU 是在轮流为多个进程服务，就好象所有的进程都在不间断地运行一样。但实际上，因为线程是 CPU 的最小运行单位，在任何一个时间内有且仅有一个线程占有 CPU。如果一台计算机有多个 CPU 或者一个 CPU 有多个核和线程，情况就不同了，如果进程产生的总线程数小于 CPU 的核数，则不同的进程的线程可以分配给不同的 CPU 来运行，这样，各个进程就是真正同时运行的，这便是并行。但如果进程分配的线程数大于 CPU 的核心线程数，则这时候就要使用并发技术。

一个 CPU 就像一条高速公路，如果有条高速公路 A 上面并排有 8 条车道，那么最大的并行车辆就是 8 辆；假设如果一辆车通过一个闸口的时间是 10ms，那么此条高速公路 A 上，1s 通过这个这个闸口的数量就是 1000ms*8/10ms。若果再复杂些，就是每辆车的发动机可能不一样处理的过程可能不一样，不一定都是 10ms，有的慢些，有的快些，这时就用到了“时间片轮转进程调

度算法”，慢的就让他退出，让下一辆先过去，一会又轮到慢的，就是这样交替切换通过闸口关卡。

而对于一个 Web 程序并发来说，1s（或者前端程序展示给用户的最大可容忍时间 3s 等）内的并发量可能计算起来就没这么简单了，影响因素可能会有：程序执行时间，CPU 的时间片轮换时间，程序占用的内存大小，程序占用的宽带大小，请求数据库的时间等等，这里就不多说了。淘宝前台项目工程师蒋江伟曾经给过系统最大并发量的算法，大家没事可以去研究研究。

这里仅仅给一个极限的并发量的算法的例子吧。

当不考虑任何客观因素的情况下，假设一个服务器有 2 个物理 CPU，每个 CPU 有 8 核 16 个线程；那么它的 1s 极限并发量是：1000ms*16(CPU 线程量)*2 个/(CPU 切片轮换时间(假设 10ms)+程序执行时间(假设 10ms))= 1600 个并发。如果前台保证不死机等情况，假设用户可容忍的最大时间是 3s，那么此套程序的最大并发量就是 4800 个。当然了这是一种理想状况，实际上加上其他各种条条框框肯定要比这个少很多。

1.7 什么是吞吐量

吞吐量是指对网络、设备、端口、虚电路或其他设施，单位时间内成功地传送数据的数量（以比特、字节、分组等测量）。

- 网络吞吐量：网络吞吐量是指在某个时刻，在网络中的两个节点之间，提供给网络应用的剩余带宽。即在没有帧丢失的情况下，设备能够接受的最大速率。网络吞吐量可以帮组寻找网络路径中的瓶颈。
- 系统吞吐量：系统吞吐量是指系统在单位时间内所处理的信息量，它以每个时间段所处理的进程数来度量。

1. 网络吞吐量

网络吞吐量是指在某个时刻，在网络中的两个节点之间，提供给网络应用的剩余带宽。即在没有帧丢失的情况下，设备能够接受的最大速率。

（1）吞吐量的大小主要由防火墙内网卡，及程序算法的效率决定，尤其是程序算法，会使防火墙系统进行大量运算，通信量大打折扣。因此，大多数防火墙虽号称 100MB 防火墙，由于其算法依靠软件实现，通信量远远没有达到 100MB，实际只有 10MB~20MB。纯硬件防火墙，由于采用硬件进行运算，因此吞吐量可以达到线性 90MB~95MB，是真正的 100MB 防火墙。

（2）吞吐量和报文转发率是关系防火墙应用的主要指标，一般采用 FDT（Full Duplex Throughput）来衡量，指 64B 数据包的全双工吞吐量，该指标既包括吞吐量指标也涵盖了报文转发率指标。

（3）吞吐量的测试方法：在测试中以一定速率发送一定数量的帧，并计算待测设备传输的帧，

如果发送的帧与接收的帧数量相等，那么就将发送速率提高并重新测试；如果接收帧少于发送帧，则降低发送速率重新测试，直至得出最终结果。吞吐量测试结果以 bit/s 或 B/s 表示。

那这里的吞吐量与带宽有什么区别呢。

吞吐量和带宽是很容易搞混的一个词，两者的单位都是 Mbit/s。先让我们来看两者对应的英语，吞吐量是 throughput，带宽是 Max net bitrate。当我们讨论通信链路的带宽时，一般是指链路上每秒所能传送的比特数。我们可以说以太网的带宽是 10Mbit/s。但是，我们需要区分链路上的可用带宽（带宽）与实际链路中每秒所能传送的比特数（吞吐量）。我们倾向于用“吞吐量”一次来表示一个系统的测试性能。这样，因为实现受各种低效率因素的影响，所以由一段带宽为 10Mbit/s 的链路连接的一对节点可能只能达到 2Mbit/s 的吞吐量。这样就意味着，一个主机上的应用能够以 2Mbit/s 的速度向另外的一个主机发送数据。

2. 系统吞吐量

系统吞吐量是指系统在单位时间内所处理的信息量，它以每小时或每天所处理的进程数来度量。

影响吞吐量因素有：

（1）存储设备的存取速度，即从存储器读出数据或数据写入存储器所需时间。

（2）CPU 性能，即 CPU 的如下 3 个参数：

- 时钟频率。
- 每条指令所花的时钟周期数（即 CPI）。
- 指令条数。

（3）系统结构，如并行处理结构可增大吞吐量。

1.8 高并发编程的意义及其好处和注意事项

由于多核多线程的 CPU 的诞生，多线程、高并发的编程越来越受重视和关注。多线程可以给程序带来如下好处。

（1）充分利用 CPU 的资源

从上面的 CPU 的介绍，可以看的出来，现在市面上没有 CPU 的内核不使用多线程并发机制的，特别是服务器还不止一个 CPU，如果还是使用单线程的技术做思路，明显就 out 了。因为程序的基本调度单元是线程，并且一个线程也只能在一个 CPU 的一个核的一个线程跑，如果你是一个 i3 的 CPU 的话，最差也是双核心 4 线程的运算能力；如果是一个线程的程序的话，那是要浪费 3/4 的 CPU 性能；如果设计一个多线程的程序的话，那它就可以同时在多个 CPU 的多个核的多个线程上跑，可以充分地利用 CPU，减少 CPU 的空闲时间，发挥它的运算能力，提高并发量。

就像我们平时坐地铁一样，很多人坐长线地铁的时候都在认真看书，而不是为了坐地铁而坐地铁，到家了再去看书，这样你的时间就相当于有了两倍。这就是为什么有些人时间很充裕，而有些人老是说没时间的一个原因，工作也是这样，有的时候可以并发地去做几件事情，充分利用我们的时间，CPU 也是一样，也要充分利用。

（2）加快响应用户的时间

比如我们经常用的迅雷下载，都喜欢多开几个线程去下载，谁都不愿意用一个线程去下载，为什么呢？答案很简单，就是多个线程下载快啊。

我们在做程序开发的时候更应该如此，特别是我们做互联网项目，网页的响应时间若提升 1s，如果流量大的话，就能增加不少转换量。做过高性能 Web 前端调优的都知道，要将静态资源地址用两三个子域名去加载，为什么？因为每多一个子域名，浏览器在加载你的页面的时候就会多开几个线程去加载你的页面资源，提升网站的响应速度。多线程，高并发真的是无处不在。

（3）可以使你的代码模块化，异步化，简单化

例如我们在做 Android 程序开发的时候，主线程的 UI 展示部分是一块主代码程序部分，但是 UI 上的按钮用相应事件的处理程序就可以做个单独的模块程序拿出来。这样既增加了异步的操作，又使程序模块化，清晰化和简单化。

时下最流行的异步程序处理机制，正是多线程、并发程序最好的应用例子。

相信多线程应用开发的好处还有很多，大家在日后的代码编写过程中可以慢慢体会它的魅力所在。

接下来谈谈高并发、多线程要注意的问题：

（1）线程之间的安全性

从前面的章节中我们都知道，在同一个进程里面的多线程是资源共享的，也就是都可以访问同一个内存地址当中的一个变量。例如：若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则就可能影响线程安全。

这个我们会在后面章节中详细的说明。

（2）线程之间的死循环过程

为了解决线程之间的安全性引入了 Java 的锁机制，而一不小心就会产生 Java 线程死锁的多线程问题，因为不同的线程都在等待那些根本不可能被释放的锁，从而导致所有的工作都无法完成。假设有两个线程，分别代表两个饥饿的人，他们必须共享刀叉并轮流吃饭。他们都需要获得两个锁：共享刀和共享叉的锁。

假如线程 A 获得了刀，而线程 B 获得了叉。线程 A 就会进入阻塞状态来等待获得叉，而线程 B 则阻塞来等待线程 A 所拥有的刀。这只是人为设计的例子，但尽管在运行时很难探测到，这类情况却时常发生。

（3）线程太多了会将服务器资源耗尽形成死机当机

线程数太多有可能造成系统创建大量线程而导致消耗完系统内存以及 CPU 的“过渡切换”，造成系统的死机，那么我们该如何解决这类问题呢？

某些系统资源是有限的，如文件描述符。多线程程序可能耗尽资源，因为每个线程都可能希望有一个这样的资源。如果线程数相当大，或者某个资源的侯选线程数远远超过了可用的资源数，则最好使用 *资源池*。一个最好的示例是数据库连接池。只要线程需要使用一个数据库连接，它就从池中取出一个，使用以后再将它返回池中。资源池也称为 *资源库*。这里先有一个概念，后面会逐渐讲到线程池的概念。

多线程应用开发的注意事项很多，希望大家在日后的工作中可以慢慢体会它的危险所在。

1.9 分布式、并行运算、并发运算

分布式计算研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给许多计算机进行处理，最后把这些计算结果综合起来得到最终的结果。最近的分布式计算项目已经被用于使用世界各地成千上万位志愿者的计算机的闲置计算能力，通过因特网，可以分析来自外太空的电讯号，寻找隐蔽的黑洞，并探索可能存在的外星智慧生命等。

并行计算与分布式计算的区别：

（1）简单的理解，并行计算借助并行算法和并行编程语言能够实现进程级并行（如 MPI）和线程级并行（如 openMP）。而分布式计算只是将任务分成小块到各个计算机分别计算各自执行。

（2）粒度方面，并行计算中，处理器间的交互一般很频繁，往往具有细粒度和低开销的特征，并且被认为是可靠的。而在分布式计算中，处理器间的交互不频繁，交互特征是粗粒度，并且被认为是不可靠的。并行计算注重短的执行时间，分布式计算则注重长的正常运行时间。

（3）联系，并行计算和分布式计算两者是密切相关的。某些特征与程度（处理器间交互频率）有关，而我们还未对这种交叉点（crossover point）进行解释；另一些特征则与侧重点有关（速度与可靠性），而且我们知道这两个特性对并行和分布两类系统都很重要。

（4）总之，这两种不同类型的计算在一个多维空间中代表不同但又相邻的点。

并发计算与分布式计算的关系：

（1）并发在单个资源个体的情况下怎么样达到最大的利用价值。比如说：1 个服务器 4 个 CPU * 4 核，并行是 16，而高并发就可能实现 160 个，用到的技术就是多线程。

（2）分布式呢，当来 1 万个并发，以前的资源满足不了要求了，那就并行的再多开几个资源服务器。而这种各个服务器之间就叫做分布式，说白了就是，不是一台 Server 能干得了得计算了，多找几个服务器协作完成。常见的架构有 Hadoop、Hbase、Zookeeper、Redis 等。

1.10 Linux和Windows对于并发采取的不同机制

Linux 可以通过多进程实现并发操作，因为在 Linux 操作系统中有一个子进程的概念，子进程继承了对应的父进程的大部分属性，如文件描述符。在 Unix 中，子进程通常为系统调用 fork 的产物。在此情况下，子进程一开始就是父进程的副本，而在这之后，根据具体需要，子进程可以借助 exec 调用来链式加载另一程序。一个进程可能下属多个子进程，但最多只能有 1 个父进程，而若某一进程没有父进程，则可知该进程很可能由内核直接生成。但是同时也对多线程并发支持得很好。

而在现在的 Windows 下面对子进程支持得不是太好，大部分都是采用一个进程下面采用多线程并发的模式。

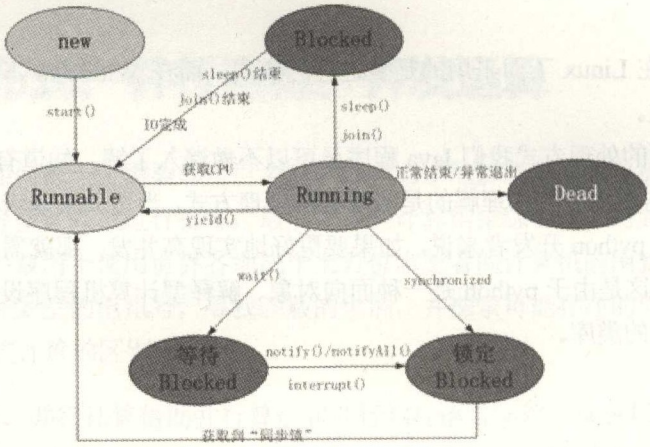
例如：Apache 在 Linux 下面采用的是多进程的模式，而在 Windows 下面采用的是多线程并发的模式，各有千秋。

但是这个多进程的处理方式我们 Java 程序员可以不做深入了解，知道有这么回事就可以了。因为我们 Java 程序需要更关注和理解的是多线程的处理方式，当然如果能清楚理解多进程的处理方式就更好了！对于 python 开发者来说，如果要更好地实现高并发，那就需要详细地理解操作系统中进程的问题了，这是由于 python 是一种面向对象、解释型计算机程序设计语言，里面分别提供了多进程和多线程的类库。

第 2 章

认识Java里面的Thread

道可顿悟，事须渐修；一切从基础做起，一点一滴地慢慢积累。



本章讲解 Java 里面 Thread 的基础。

2.1 线程简单实现的三种方法

线程是程序中的执行线程。Java 虚拟机允许应用程序并发地运行多个执行线程。每个线程都有一个优先级，高优先级线程的执行优先于低优先级线程。每个线程都可以或不可以标记为一个守护程序。当某个线程中运行的代码创建一个新 Thread 对象时，该新线程的初始优先级被设定为创建线程的优先级，并且当且仅当创建线程是守护线程时，新线程才是守护程序。当 Java 虚拟机启动时，通常都会有单个非守护线程（它通常会调用某个指定类的 main 方法）。Java 虚拟机会继续执行线程，直到下列任一情况出现时为止：调用了 Runtime 类的 exit 方法，并且安全管理器允许退出操作发生。非守护线程的所有线程都已停止运行，无论是通过从对 run 方法的调用中返回，还是通过抛出一个传播到 run 方法之外的异常。

接下来我们来慢慢揭开 Thread 的神秘面纱。先来说一下开启新线程常见的三种方法。

（1）第一种创建线程的方式是直接 extends Thread 覆盖 run()方法即可。代码和运行结果如下：

```
package demo.thread;
/**
* 第一种创建线程的方式是直接 extends Thread 覆盖 run()方法即可。
*/
public class ThreadA extends Thread {
      public void run() {
            super.run();
            try {
                  // TODO Auto-generated method stub
                  //模拟做事情执行了500ms。
                  Thread. sleep(500L);
            } catch (InterruptedException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
            System. out.println("这是线程 A" );
      }
}
```

执行线程程序如下：ThreadMain

```
package demo.thread;
public class ThreadMain {
      public static void main(String[] args) {
            ThreadA threada = new ThreadA();
            threada.start();//启动线程 A
            System. out.println("这是主线程: " );
      }
}
```

这种实现方式的缺点是，一个 Java 类只能 extend 继承一个父类；有的时候不免有点尴尬。它的执行结果如下：

```
这是主线程:
这是线程 A
```

（2）第二种实现方式是实现 Runnable 接口，实现 run()方法。具体实现如下所示：

```
package demo.thread;
//implements Runnable 接口，实现 run()方法。
```

```
//优点，Java 里面可以有多个接口，解决 extends 的缺点
public class ThreadB implements Runnable {
    public void run() {
        try {
            // TODO Auto-generated method stub
            //模拟做事情执行了500ms。
            Thread. sleep(500L);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System. out.println("这是线程 B" );
    }
}
package demo.thread;
    public class ThreadMain {
     public static void main(String[] args) {
        ThreadB threadb = new ThreadB();
         new Thread(threadb).start();//注意启动方式有点不一样。
        System. out.println("这是主线程; " );
    }
}
```

执行结果如下：

```
这是主线程;
这是线程 B
```

（3）第三种实现方式是 implements Callable，实现 call()方法可以得到线程的执行结果。具体实现如下所示。

ThreadC 的类如下：

```
package demo.thread;
    import java.util.concurrent.Callable;
    //实现 Callable 接口，实现 call()方法。
    //可以有返回结果，注意这次是要覆盖 call 方法。
    public class implements Callable<String> {
        public String call() throws Exception {
            try {
                // TODO Auto-generated method stub
                //模拟做事情执行了500ms。
```

```
                Thread. sleep(500L);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            System. out.println("这是线程 B" );
             return "thread B" ;
        }
    }
```

ThreadMain 实例如下：

```
package demo.thread;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;
public class ThreadMain {
    public static void main(String[] args) {
        ThreadC threadc = new ThreadC();
        //FutureTask 后续会讲到，先知道有这么个实现方式
        FutureTask<String> faeature = new FutureTask<String>(threadc);
        new Thread(faeature).start();//注意启动方式有点不一样。
        System. out.println("这是主线程; begin! " );
        //注意细细体会这个，只有主线程 get 了，主线程才会继续往下面执行
        try {
            System. out.println("得到的返回结果是:" +faeature.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        System. out.println("这是主线程; end! " );
    }
}
```

执行结果如下：

```
这是主线程; begin!
这是线程 B
thread B
这是主线程; end!
```

总之，前言万语，不如仔细研读代码和体会上面三种实现方式的优缺点和不同之处，比较其运行结果，并且要注意到运行和执行开启线程的方法也不尽相同。

2.2 Thread 里面的属性和方法

上一节我们讲述了线程的三种实现方式，由于线程的第二种实现方式 ThreadB 是最常见的方式，后面我们就以这种方式制做案例多一些。

第一步：先看一下 Thread 的 API 有哪些常用的方法，接下来我们找几个方法做一下测试。Thread 详细的 API 请参看附录五。

构造方法常用的有：

- Thread()分配新的 Thread 对象。
- Thread(Runnable target)分配新的 Thread 对象。
- Thread(Runnable target, String name)分配新的 Thread 对象。
- Thread(String name)分配新的 Thread 对象。
- Thread(ThreadGroup group, String name) 分配新的 Thread 对象。

主要的方法有如下表所示：

方法	说明
static int activeCount()	返回当前线程的线程组中活动线程的数目
void checkAccess()	判定当前运行的线程是否有权修改该线程
static Thread currentThread()	返回对当前正在执行的线程对象的引用
static void dumpStack()	将当前线程的堆栈跟踪打印至标准错误流
static int enumerate(Thread[] tarray)	将当前线程的线程组及其子组中的每一个活动线程复制到指定的数组中
static Map<Thread,StackTraceElement[]> getAllStackTraces()	返回所有活动线程的堆栈跟踪的一个映射
ClassLoader getContextClassLoader()	返回该线程的上下文 ClassLoader
static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()	返回线程由于未捕获到异常而突然终止时调用的默认处理程序
long getId()	返回该线程的标识符
String getName()	返回该线程的名称
int getPriority()	返回线程的优先级
StackTraceElement[] getStackTrace()	返回一个表示该线程堆栈转储的堆栈跟踪元素数组
Thread.State getState()	返回该线程的状态
ThreadGroup getThreadGroup()	返回该线程所属的线程组

（续表）

方法	说明
Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()	返回该线程由于未捕获到异常而突然终止时调用的处理程序
static boolean holdsLock(Object obj)	当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true
void interrupt()	中断线程
static boolean interrupted()	测试当前线程是否已经中断
boolean isAlive()	测试线程是否处于活动状态
boolean isDaemon()	测试该线程是否为守护线程
boolean isInterrupted()	测试线程是否已经中断
void join()	等待该线程终止
void join(long millis)	等待该线程终止的时间最长为 millis ms
void join(long millis, int nanos)	等待该线程终止的时间最长为 millis ms + nanos ns
void run()	如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回
void setContextClassLoader(ClassLoader cl)	设置该线程的上下文 ClassLoader
void setDaemon(boolean on)	将该线程标记为守护线程或用户线程
static void setDefaultUncaughtExceptionHandler (Thread.UncaughtExceptionHandler eh)	设置当线程由于未捕获到异常而突然终止，并且没有为该线程定义其他处理程序时所调用的默认处理程序
void setName(String name)	改变线程名称，使之与参数 name 相同
void setPriority(int newPriority)	更改线程的优先级
static void sleep(long millis)	在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响
static void stop()	停止线程，不推荐使用，线程不安全的
void start()	启动线程
static void yield()	暂停当前正在执行的线程对象，并执行其他线程

单独说一下线程的优先级：可以调用 Thread 类的方法 getPriority() 和 setPriority()来存取线程的优先级。线程的优先级代表该线程的重要程度，当有多个线程同时处于可执行状态并等待获得 CPU 时间时，线程调度系统根据各个线程的优先级来决定给谁分配 CPU 时间，优先级高的线程有更大的机会获得 CPU 时间，优先级低的线程也不是没有机会，只是机会要小一些罢了。你可以调用 Thread 类的方法 getPriority() 和 setPriority()来存取线程的优先级，线程的优先级界于 1(MIN_PRIORITY)和 10(MAX_PRIORITY)之间，缺省是 5(NORM_PRIORITY)。

第二步：我们新建 5 个线程实例来体会一下各个线程的属性：

（1）ThreadB 的内容修改成如下：

```
package demo.thread;
```

```
public class ThreadB implements Runnable {
        public void run() {
                try {
                        //模拟做事情执行了100s，以便一会我们的监控工具监控到！
                        Thread. sleep(100000L);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                Thread curThread = Thread. currentThread() ;
                String curThreadName = curThread.getName();
                System. out.println("这是线程的名称:" +curThread.getName());
                System. out.println("返回当前线程" +curThreadName+"的线程组中活动线程的数目: " +Thread.activeCount ());
                System. out.println(" 返 回 该 线 程 " +curThreadName+" 的 标 识 符 : "+curThread.getId());
                System. out.println(" 返 回 线 程 " +curThreadName+" 的 优 先 级:"+curThread.getPriority());
                System. out.println(" 返 回 该 线 程 " +curThreadName+" 的 状 态:"+curThread.getState());
                System. out.println(" 返 回 该 线 程 " +curThreadName+" 所 属 的 线 程 组:"+curThread.getThreadGroup());
                System. out.println(" 测 试 线 程 " +curThreadName+" 是 否 处 于 活 动 状 态:"+curThread.isAlive());
                System. out.println("测试线程" +curThreadName+"是否测试该线程是否为守护线程:" +curThread.isDaemon());
        }
}
```

（2）执行线程 ThreadMain 的内容如下：

```
package demo.thread;
public class ThreadMain {
        public static void main(String[] args) {
                ThreadB threadb = new ThreadB();
                for(int i=0;i<5;i++) {
                        new Thread(threadb,"线程名称：(" +i+")").start();
                }
                //返回对当前正在执行的线程对象的引用。此处获得我们的主线程
                Thread threadMain = Thread. currentThread() ;
                System. out.println("这是主线程：" );
```

```
                System.out.println("返回当前线程的线程组中活动线程的数目："
+Thread.activeCount ());
                System.out.println("主线程的名称：" +threadMain.getName());
                System.out.println("返回该线程的标识符：" +threadMain.getId());
                System.out.println("返回线程的优先级:" +threadMain.getPriority());
                System.out.println("返回该线程的状态:" +threadMain.getState());
                System.out.println("返回该线程所属的线程
组:" +threadMain.getThreadGroup());
                System.out.println("测试该线程是否为守护线程:" +threadMain.isAlive());
                try {
                      Thread. sleep(10000L);//休息10s 以便我们的监控工具能监控到
                } catch (InterruptedException e) {
                      e.printStackTrace();
                }
          }
    }
```

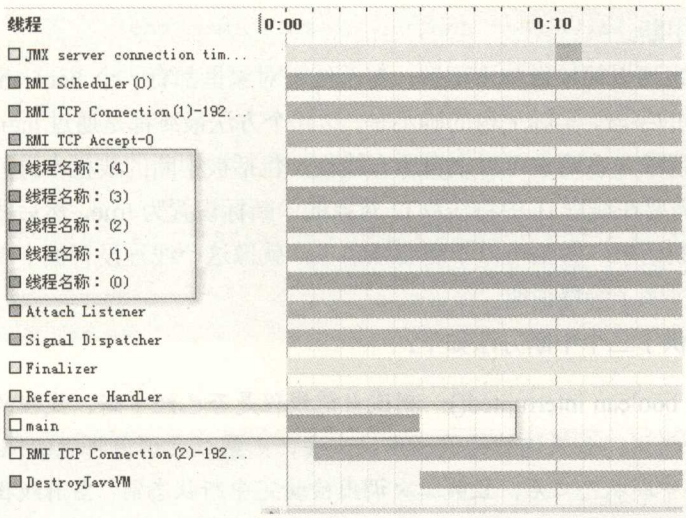
执行的结果如下所示，请仔细体会执行结果，你会有所发现！结合第 1 章所讲的仔细体会一下！

```
这是主线程。
返回当前线程的线程组中活动线程的数目：6
主线程的名称：main
返回该线程的标识符：1
返回线程的优先级:5
返回该线程的状态:RUNNABLE
返回该线程所属的线程组:java.lang.ThreadGroup[name=main,maxpri=10]
测试该线程是否为守护线程:true
这是线程的名称:线程名称：(0)
返回当前线程名称：(0)的线程组中活动线程的数目：6
返回该线程名称：(0)的标识符：8
返回线程名称：(0)的优先级:5
这是线程的名称:线程名称：(1)
这是线程的名称:线程名称：(2)
返回当前线程名称：(2)的线程组中活动线程的数目：6
返回该线程名称：(2)的标识符：10
返回线程名称：(2)的优先级:5
返回该线程名称：(2)的状态:RUNNABLE
这是线程的名称:线程名称：(3)
返回当前线程名称：(3)的线程组中活动线程的数目：6
```

```
返回该线程名称：(3)的标识符：11
返回线程名称：(3)的优先级:5
返回该线程名称：(3)的状态:RUNNABLE
返回该线程名称：(2)所属的线程组:java.lang.ThreadGroup[name=main,maxpri=10]
返回当前线程名称：(1)的线程组中活动线程的数目：6
返回该线程名称：(1)的标识符：9
返回线程名称：(1)的优先级:5
返回该线程名称：(1)的状态:RUNNABLE
这是线程的名称:线程名称：(4)
返回当前线程名称：(4)的线程组中活动线程的数目：6
返回该线程名称：(4)的标识符：12
返回线程名称：(4)的优先级:5
返回该线程名称：(4)的状态:RUNNABLE
返回该线程名称：(0)的状态:RUNNABLE
返回该线程名称：(4)所属的线程组:java.lang.ThreadGroup[name=main,maxpri=10]
测试线程名称：(4)是否处于活动状态:true
返回该线程名称：(1)所属的线程组:java.lang.ThreadGroup[name=main,maxpri=10]
测试线程名称：(1)是否处于活动状态:true
测试线程名称：(1)是否测试该线程是否为守护线程:false
测试线程名称：(2)是否处于活动状态:true
测试线程名称：(2)是否测试该线程是否为守护线程:false
返回该线程名称：(3)所属的线程组:java.lang.ThreadGroup[name=main,maxpri=10]
测试线程名称：(4)是否测试该线程是否为守护线程:false
返回该线程名称：(0)所属的线程组:java.lang.ThreadGroup[name=main,maxpri=10]
测试线程名称：(0)是否处于活动状态:true
测试线程名称：(0)是否测试该线程是否为守护线程:false
测试线程名称：(3)是否处于活动状态:true
测试线程名称：(3)是否测试该线程是否为守护线程:false
```

第三步，看一下我们监控的结果，因为我们设置了时间戳会看到不同的监控结果。

在上面 5 个线程和一个 main 主线程中，我们可以发现主线程先于另外 5 个线程先结束了，如下图所示；大家仔细看看代码就能体会出来了，这是时间上控制的，这里先有个印象，后面会讲线程的具体监控方法。



Thread 使用时的注意事项如下：

（1）开启一个新的线程的时候，一定要给它一个名字，通过上面的图我们也看到了，方便我们跟踪观察线程，对号入座，方便排查问题。否则监控的时候就没有办法很直观地知道某个线程的用途。

（2）需要注意的是，resume、stop、suspend 等方法已经被废除，不建议使用，建议使用信号量（共享变量）或 interrupt 方法来代替 stop 方法等。

（3）main 方法主线程结束了，新开启的子线程不一定结束。

2.3 关于线程的中断机制

要使任务和线程能安全、快速、可靠地停止下来，并不是一件容易的事。Java 没有提供任何机制来安全地终止线程，那么我们又该如何使用线程的停止或者中断呢？

1. 第一种方式：调用 Thread.stop()

该方法强迫停止一个线程，并抛出一个新创建的 ThreadDeath 对象作为异常。停止一个尚未启动的线程是允许的，如果稍后启动该线程，它会立即终止。通常不应试图捕获 ThreadDeath，除非它必须执行某些异常的清除操作。如果 catch 子句捕获了一个 ThreadDeath 对象，则必须重新抛出该对象，这样该线程才会真正终止。

然而，Thread.stop()不安全，已不再建议使用，这意味着在未来的 JAVA 版本中，它将不复存在。所以我们也不多说了。

2. 第二种方式：利用 Thread.interrupt()方法和机制

Java 中断机制是一种协作机制，也就是说通过中断并不能直接终止另一个线程，而需要被中

断的线程自己处理中断。

Java 中断模型也可以简单做如下理解，每个线程对象里都有一个 boolean 类型的标识（不一定就要是 Thread 类的字段，实际上也的确不是，这几个方法最终都是通过 native 方法来完成的），代表着是否有中断请求（该请求可以来自所有线程，包括被中断的线程本身）。例如，当线程 t1 想中断线程 t2，只需要在线程 t1 中将线程 t2 对象的中断标识置为 true，然后线程 2 可以选择在合适的时候处理该中断请求，甚至可以不理会该请求，就像这个线程没有被中断一样。

接下来我们通过例子慢慢说明。

Thread 类中提供了三个中断方法如下：

- public static boolean interrupted(): 测试当前线程是否已经中断。线程的中断状态由该方法清除。换句话说，如果连续两次调用该方法，则第二次调用将返回 false（在第一次调用已清除了其中断状态之后，且第二次调用检验完中断状态前，当前线程再次中断的情况除外）。
- public boolean isInterrupted(): 测试线程是否已经中断。线程的中断状态不受该方法的影响。
- public void interrupt(): 中断线程，但是没有返回结果。是唯一能将中断状态设置为 true 的方法。

上面的例子中，线程 t1 通过调用 interrupt 方法将线程 t2 的中断状态置为 true，t2 可以在合适的时候调用 interrupted 或 isInterrupted 来检测状态并做相应的处理。

我们先来看看下面的实例。

需求：我们用 main 线程 1 来中断 InterruptDemo 线程 2，代码如下：

```
public class ThreadInterruptDemo implements Runnable {
        public static void main(String[] args) throws Exception {
                Thread thread = new Thread
                (new ThreadInterruptDemo(), "InterruptDemo Thread");
                System. out.println("Starting thread..." );
                thread.start();
                Thread. sleep(3000);
                System. out.println("Interrupting thread..." );
                thread.interrupt();
                System. out.println("线程是否中断: " + thread.isInterrupted());
                Thread. sleep(3000);
                System. out.println("Stopping application..." );
        }
        public void run() {
                boolean stop = false ;
                while (!stop) {
                        System. out.println("My Thread is running..." );
```

```
                    long time = System.currentTimeMillis();
                    while ((System.currentTimeMillis() - time < 1000)) {
                        // 让该循环持续一段时间，使用上面的话打印次数少点
                    }
                }
                System. out.println("My Thread exiting under request..." );
        }
}
```

运行结果如下：

```
Starting thread...
My Thread is running...
My Thread is running...
Interrupting thread...
线程是否中断：true
My Thread is running...
My Thread is running...
Stopping application...
My Thread is running...
My Thread is running...
My Thread is running...
......
```

从运行结果上看到，InterruptDemo 线程 2 永远没有结束。就像我们前面所描述的一样，线程 2 没有处理该中断请求，就像这个线程没有被中断一样。接下来，我们将上面的例子稍作改进，让线程 2 处理终止请求。

添加如下代码：

```
if (Thread.currentThread().isInterrupted()) {
    // 需要线程本身去处理一下它的终止状态
    break;
}
```

代码就变成了如下内容：

```
public void run() {
            boolean stop = false ;
            while (!stop) {
                System. out.println("My Thread is running..." );
                long time = System.currentTimeMillis();
```

```
                while ((System.currentTimeMillis() - time < 1000)) {
                    // 让该循环持续一段时间，使用上面的话打印次数少点
                }
                if (Thread.currentThread().isInterrupted()) {
                    // Thread.currentThread()取得当前线程
                    // 需要线程本身去处理一下它的终止状态
                    break;
                }
            }
            System.out.println("My Thread exiting under request..." );
        }
```

运行结果如下：

```
Starting thread...
My Thread is running...
My Thread is running...
My Thread is running...
My Thread is running...
Interrupting thread...
线程是否中断：true
My Thread exiting under request...
Stopping application...
```

从运行结果上看到，InterruptDemo 线程 2 真正的终止结束了。就像我们前面所描述的一样，线程 2 处理了该中断请求，从而就会真正的中断。

其实 interrupt 还可以处理一些更为复杂的逻辑，当外部线程对某线程调用了 thread.interrupt() 方法后，Java 语言的处理机制是这样的：

如果该线程处在可中断状态下（调用了 Thread.wait()或者 Thread.sleep()等特定会发生阻塞的 api），那么该线程会立即被唤醒，同时会受到一个 InterruptedException，同时，如果是阻塞在 IO 上，对应的资源会被关闭。如果该线程接下来不执行 Thread.interrupted()方法（不是 interrupt），那么该线程处理任何 IO 资源的时候，都会导致这些资源关闭。当然，解决的办法就是调用一下 interrupted()，不过这里需要程序员自行根据代码的逻辑来设定，根据自己的需求确认是否可以直接忽略该中断，还是应该马上退出。

简单的异常处理方式如下：

```
try {
    Thread.sleep(3L);
} catch (InterruptedException e) {
    break;
```

```
}
```

以上面的 demo 为例之后代码就变成了如下内容：

```
public void run() {
            boolean stop = false ;
            while (!stop) {
                  System. out.println("My Thread is running..." );
                 try {
                        Thread.sleep(3L);
                  } catch (InterruptedException e) {
                        break;
                  }
            }
            System. out.println("My Thread exiting under request..." );
}
```

运行完之后，运行结果可以达到不变的效果。

最后，需要注意的是，被终止的线程一定要对 isInterrupted 状态进行处理，否则如果代码是死循环的情况下，线程将永远都不会结束。

2.4 线程的生命周期

线程的生命周期一共有 5 个状态：new 、runnable、running、blocked、dead。

1. 线程生命周期的 5 种状态

（1）新建（new Thread）：当创建 Thread 类的一个实例（对象）时，此线程进入新建状态（未被启动），也就说处于新生状态的线程有自己的内存空间，但该线程并没有运行。此时线程还不是活着的（not alive）。例如：

```
Thread  t1=new Thread();
```

（2）就绪（runnable）：线程已经被启动，正在等待被分配给 CPU 时间片。也就通过调用线程实例的 start()方法来启动线程使线程进入就绪状态（runnable）。处于就绪状态的线程已经具备了运行条件，但还没有被分配到 CPU，不一定会被立即执行，此时处于线程就绪队列，等待系统为其分配 CPU，等待状态并不是执行状态。此时线程是活着的（alive）。

（3）运行（running）：线程获得 CPU 资源正在执行任务（run()方法），此时除非此线程自动放弃 CPU 资源或者有优先级更高的线程进入，线程将一直运行到结束。此时线程是活着的

（alive）。

（4）堵塞（blocked）：由于某种原因导致正在运行的线程让出 CPU 并暂停自己的执行，即进入堵塞状态。

- 正在睡眠：用 sleep(long t) 方法可使线程进入睡眠方式。一个睡眠着的线程在指定的时间过去可进入就绪状态。
- 正在等待：调用 wait()方法。可调用 notify()方法回到就绪状态
- 被另一个线程所阻塞：调用 suspend()方法。可调用 resume()方法恢复。

处于 Blocking 状态的线程仍然是活着的（alive）。

（5）死亡（dead）：当线程执行完毕或被其他线程杀死，线程就进入死亡状态，这时线程不可能再进入就绪状态等待执行。

- 自然终止：正常运行 run()方法后终止。
- 异常终止：调用 stop()方法让一个线程终止运行。

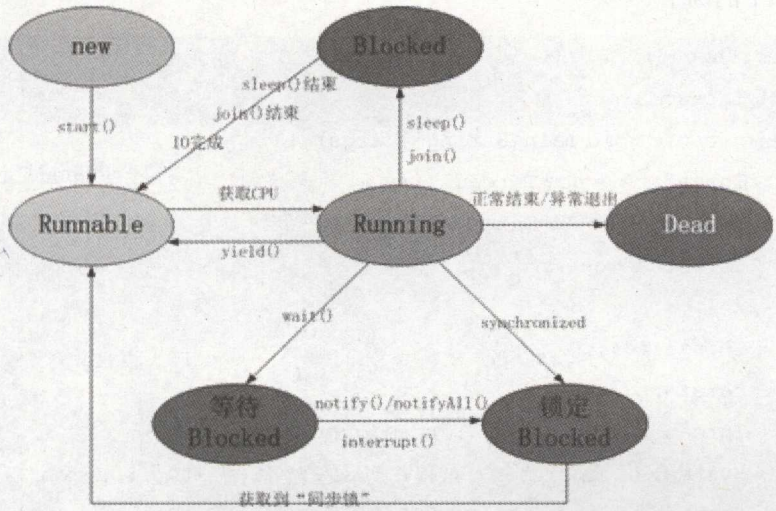
处于 Dead 状态的线程不是活着的（not alive）。

2. 线程常用方法

与线程状态对应的常用方法有：

- void run()：创建该类的子类时必须实现的方法。
- void start()：开启线程的方法。
- static void sleep(long t) / static void sleep(long millis,int nanos)：释放 CPU 的执行权，不释放锁。当前线程睡眠/millis 的时间（millis 指定睡眠时间是其最小的不执行时间，因为 sleep(millis)休眠到达后，无法保证会被 JVM 立即调度），sleep()是一个静态方法（static method），所以它不会停止其他的线程也处于休眠状态。线程 sleep()时不会失去拥有的对象锁。作用是：保持对象锁，让出 CPU，调用目的是不让当前线程独自霸占该进程所获取的 CPU 资源，以留一定的时间给其他线程执行的机会。
- final void wait()：释放 CPU 的执行权，释放锁。当一个线程执行到 wait()方法时，它就进入到一个和该对象相关的等待池（Waiting Pool）中，同时失去了对象的机锁—暂时的，wait 后还要返还对象锁。当前线程必须拥有当前对象的锁，如果当前线程不是此锁的拥有者，会抛出 IllegalMonitorStateException 异常，所以 wait()必须在 synchronized block 中调用。
- final void notify()/notifyAll()：唤醒在当前对象等待池中等待的第一个线程/所有线程。notify()/notifyAll()也必须拥有相同对象锁，否则也会抛出 IllegalMonitorStateException 异常。
- static void yied()：可以对当前线程进行临时暂停，让出 CPU 的使用权，给其他线程执行机会、让同等优先权的线程运行（但并不保证当前线程会被 JVM 再次调度、使该线程重新进入 Running 状态），如果没有同等优先权的线程，那么 yield()方法将不会起作用。

用一张图来表示的话就是：



2.5 什么是守护线程

（1）守护线程，可以简单地理解为后台运行线程。进程结束，守护线程自然而然地就会结束，不需要手动的去关心和通知其状态。例如：在你的应用程序运行时播放背景音乐，在文字编辑器里做自动语法检查、自动保存等功能。Java 的垃圾回收也是一个守护线程。守护线程的好处就是你不需要关心它的结束问题。例如你在你的应用程序运行的时候希望播放背景音乐，如果将这个播放背景音乐的线程设定为非守护线程，那么在用户请求退出的时候，不仅要退出主线程，还要通知播放背景音乐的线程退出；如果设定为守护线程则不需要了。

（2）守护线程与普通线程写法上基本没啥区别，调用线程对象的方法 setDaemon(true)，则可以将其设置为守护线程。该方法必须在启动线程前调用：

```
public final void setDaemon(boolean on)//该方法必须在启动线程前调用
```

将该线程标记为守护线程或用户线程。当正在运行的线程都是守护线程时，Java 虚拟机退出。该方法首先调用该线程的 checkAccess 方法，且不带任何参数。这可能抛出 SecurityException（在当前线程中）。

- 参数：on - 如果为 true，则将该线程标记为守护线程。
- 抛出：IllegalThreadStateException - 如果该线程处于活动状态。 SecurityException - 如果当前线程无法修改该线程。

（3）守护线程主动程序去使用的情况较少，但在，JVM 的垃圾回收、内存管理等线程都是守护线程。还有就是在做数据库应用时候，使用的数据库连接池，连接池本身也包含着很多后台

线程，监控连接个数、超时时间、状态等等使用情况。

（4）体会如下的例子：

```
package demo.thread;
public class ThreadMain {
        public static void main(String[] args) {
                Thread tA = new ThreadA();
                Thread tB = new ThreadB();
                tA.setDaemon(true); // 设置为守护线程,注意一定要在开始之前调用

                tB.start();
                tA.start();
                Thread mainThread = Thread.currentThread();
                System.out.println("线程A是不是守护线程" +tA.isDaemon());
                System.out.println("线程b是不是守护线程" +tB.isDaemon());
                System.out.println("线程main是不是守护线程" +mainThread.isDaemon());
        }
}
package demo.thread;
public class ThreadB extends Thread {
        public void run () {
                for (int i = 0; i < 5; i++) {
                        System.out.println("线程B第" + i + "次执行!");
                        try {
                                Thread.sleep(7);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
}
package demo.thread;
public class ThreadA extends Thread {
        public void run () {
                for(long i = 0; i < 9999999L; i++) {
                        System.out.println("后台线程A第" + i + "次执行!");
                        try {
                                Thread.sleep(7);
                        } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }
            }
        }
    }
```

执行 ThreadMain 的结果：

```
线程 B 第0次执行!
线程 A 是不是守护线程 true
线程 b 是不是守护线程 false
线程 main 是不是守护线程 false
后台线程 A 第0次执行!
线程 B 第1次执行!
后台线程 A 第1次执行!
线程 B 第2次执行!
后台线程 A 第2次执行!
线程 B 第3次执行!
后台线程 A 第3次执行!
线程 B 第4次执行!
后台线程 A 第4次执行!
后台线程 A 第5次执行!
```

从上面的执行结果可以看出：前台线程是保证执行完毕的，后台线程还没有执行完毕就退出了。

需要注意的是：

（1）JRE 判断程序是否执行结束的标准是所有的前台执线程行完毕了，而不管后台线程的状态，因此，在使用后台线程的时候一定要注意这个问题。

（2）当进程中所有非守护线程已结束或退出时，即使仍有守护线程在运行，进程仍将结束。也就是说当程序结束了，守护线程有可能依然还未退出。

2.6 线程组

在 Java 的多线程处理中有线程组 ThreadGroup 的概念，ThreadGroup 是为了方便线程管理出现的。我们可以统一设定线程组的一些属性，比如 setDaemon，设置未处理异常的处理方法，设置统一的安全策略等等，也可以通过线程组方便地获得线程的一些信息。

每一个 ThreadGroup 都可以包含一组的子线程和一组子线程组，在一个进程中线程组是以树

形的方式存在，通常情况下根线程组是 system 线程组。system 线程组下是 main 线程组，默认情况下第一级应用自己的线程组是通过 main 线程组创建出来的。也就是说 system 线程组是所有线程最顶级的父线程组。

```
Thread.currentThread().getThreadGroup();//可以获得当前线程的线程组。
```

Java 允许我们对一个线程组中的所有线程同时进行操作，比如我们可以通过调用线程组的相应方法来设置其中所有线程的优先级，也可以启动或阻塞其中的所有线程。

Java 的多线程组机制的另一个重要作用是线程安全。线程组机制允许我们通过分组来区分有不同安全特性的线程，对不同组的线程进行不同的处理，还可以通过线程组的分层结构来支持不对等安全措施的采用。Java 的 ThreadGroup 类提供了大量的方法来方便我们对线程组树中的每一个线程组以及线程组中的每一个线程进行操作。

线程组和线程池的区别：线程组和线程池是两个不同的概念，他们的作用完全不同，前者是为了方便线程的管理，后者是为了管理线程的生命周期，复用线程，减少创建销毁线程的开销。

2.7 当前线程副本：ThreadLocal

（1）当使用 ThreadLocal 维护变量时，ThreadLocal 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其他线程所对应的副本。从线程的角度看，目标变量就像是线程的本地变量，这也是类名中 Local 所要表达的意思。

（2）ThreadLocal<T>类接口很简单，只有 4 个方法：

- void set(T value)，设置当前线程的线程局部变量的值。
- public T get()，该方法返回当前线程所对应的线程局部变量。
- public void remove()，将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是 JDK 5.0 新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。
- protected T initialValue()，返回该线程局部变量的初始值，该方法是一个 protected 的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第 1 次调用 get()或 set(Object)时才执行，并且仅执行 1 次。ThreadLocal 中的缺省实现直接返回一个 null。

（3）简单的使用方法如下：

```
package demo.thread;
public class ThreadMain {
    // 1：通过匿名内部类覆盖 ThreadLocal 的 initialValue()方法，指定初始值
    private static ThreadLocal<Integer> seqNum = new ThreadLocal<Integer>() {
```

```
            public Integer initialValue() {
                return 0;
            }
        };
        public ThreadLocal<Integer> getThreadLocl() {
            return seqNum ;
        }
        // 2：获取下一个序列值
        public int getNextNum() {
            seqNum.set(seqNum .get() + 1);
            return seqNum .get();
        }
        public static void main(String[] args) {
            ThreadMain sn = new ThreadMain();
            // 3： 3个线程共享 sn，各自产生序列号
            TestClient t1 = new TestClient(sn);
            TestClient t2 = new TestClient(sn);
            TestClient t3 = new TestClient(sn);
            t1.start();
            t2.start();
            t3.start();
        }
        private static class TestClient extends Thread {
            private ThreadMain sn ;
            public TestClient(ThreadMain sn) {
                this.sn = sn;
            }
            public void run() {
                for (int i = 0; i < 3; i++) {
                    //4：每个线程打出3个序列值
                    System. out.println("thread[" +
Thread.currentThread().getName() + "] --> sn[" + sn.getNextNum() + "]");
                }
                sn.getThreadLocl().remove(); // 每个线程用完的时候要记得删除
            }
        }
    }
```

通常我们通过匿名内部类的方式来定义 ThreadLocal 的子类，提供初始的变量值，如例子中

注释 1 处所示。TestClient 线程产生一组序列号，在注释 3 处，我们生成 3 个 TestClient，它们共享同一个 TestNum 实例。运行以上代码，在控制台上输出以下的结果：

```
thread[Thread-0] --> sn[1]
thread[Thread-1] --> sn[1]
thread[Thread-2] --> sn[1]
thread[Thread-1] --> sn[2]
thread[Thread-0] --> sn[2]
thread[Thread-1] --> sn[3]
thread[Thread-2] --> sn[2]
thread[Thread-0] --> sn[3]
thread[Thread-2] --> sn[3]
```

考察输出的结果信息，我们发现每个线程所产生的序号虽然都共享同一个 TestNum 实例，但它们并没有发生相互干扰的情况，而是各自产生独立的序列号，这是因为我们通过 ThreadLocal 为每一个线程提供了单独的副本。

（4）ThreadLocal 的实现原理：那么到底 ThreadLocal 类是如何实现这种“为每个线程提供不同的变量拷贝”的呢？

先来看一下 ThreadLocal 的 set()方法的源码是如何实现的：

```
public void set(T paramT)
{
   Thread localThread = Thread.currentThread();
   ThreadLocalMap localThreadLocalMap = getMap(localThread);
   if (localThreadLocalMap != null)
     localThreadLocalMap.set( this, paramT);
   else
     createMap(localThread, paramT);
}
```

在这个方法内部我们看到，首先通过 getMap(Thread t)方法获取一个和当前线程相关的 ThreadLocalMap，然后将变量的值设置到这个 ThreadLocalMap 对象中，当然如果获取到的 ThreadLocalMap 对象为空，就通过 createMap 方法创建。

线程隔离的秘密，就在于 ThreadLocalMap 这个类。ThreadLocalMap 是 ThreadLocal 类的一个静态内部类，它实现了键值对的设置和获取（对比 Map 对象来理解），每个线程中都有一个独立的 ThreadLocalMap 副本，它所存储的值，只能被当前线程读取和修改。ThreadLocal 类通过操作每一个线程特有的 ThreadLocalMap 副本，从而实现了变量访问在不同线程中的隔离。因为每个线程的变量都是自己特有的，完全不会有并发错误。还有一点就是，ThreadLocalMap 存储的键值对中的键是 this 对象指向的 ThreadLocal 对象，而值就是你所设置的对象了。

为了加深理解，我们接着看上面代码中出现的 getMap 和 createMap 方法的实现：

```
ThreadLocalMap getMap(Thread paramThread)
{
  return paramThread. threadLocals;
}
void createMap(Thread paramThread, T paramT)
{
  paramThread.threadLocals = new ThreadLocalMap(this, paramT);
}
```

接下来再看一下 ThreadLocal 类中的 get()方法：

```
public T get()
{
  Thread localThread = Thread. currentThread();
  ThreadLocalMap localThreadLocalMap = getMap(localThread);
  if (localThreadLocalMap != null)
  {
    ThreadLocalMap.Entry localEntry = localThreadLocalMap.getEntry(this );
    if (localEntry != null)
    {
      Object localObject = localEntry. value;
      return localObject;
    }
  }
  return setInitialValue();
}
```

再来看 setInitialValue()方法：

```
private T setInitialValue()
{
  Object localObject = initialValue();
  Thread localThread = Thread. currentThread();
  ThreadLocalMap localThreadLocalMap = getMap(localThread);
  if (localThreadLocalMap != null)
    localThreadLocalMap.set( this, localObject);
  else
    createMap(localThread, localObject);
  return localObject;
```

```
}
```

获取和当前线程绑定的值时，ThreadLocalMap 对象是以 this 指向的 ThreadLocal 对象为键进行查找的，这当然和前面 set()方法的代码是相呼应的。

进一步地，我们可以创建不同的 ThreadLocal 实例来实现多个变量在不同线程间的访问隔离，为什么可以这么做？因为不同的 ThreadLocal 对象作为不同键，当然也可以在线程的 ThreadLocalMap 对象中设置不同的值了。通过 ThreadLocal 对象，在多线程中共享一个值和多个值的区别，就像你在一个 HashMap 对象中存储一个键值对和多个键值对一样，仅此而已。

（5）ThreadLocal 在处理线程的局部变量的时候比后面要讲的 synchronized 同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。

（6）注意事项：使用 ThreadLocal ，一般都是声明在静态变量中，如果不断地创建 ThreadLocal 而且没有调用其 remove 方法，将会导致内存泄露，特别是在高并发的 Web 容器当中这么做的时候。

2.8 线程异常的处理

run()方法不允许 throw exception，所有的异常必须在 run 方法内进行处理。

在 Java 多线程程序中，所有线程都不允许抛出未捕获的 checked exception，也就是说各个线程需要自己把自己的 checked exception 处理掉。这一点是通过 java.lang.Runnable.run()方法声明（因为此方法声明上没有 throw exception 部分）进行了约束。但是线程依然有可能抛出 unchecked exception，当抛出此类异常时，线程就会终结，而对于主线程和其他线程完全不受影响，且完全感知不到某个线程抛出的异常，也是说完全无法 catch 到这个异常。JVM 的这种设计源自于这样一种理念：“线程是独立执行的代码片断，线程的问题应该由线程自己来解决，而不要委托到外部。”基于这样的设计理念，在 Java 中，线程方法的异常（无论是 checked 还是 unchecked exception），都应该在线程代码边界之内（run 方法内）进行 try catch 并处理掉。

因此，在 thread 里面，如果要处理 checked exception，简单的一个 try/catch 块就可以了。对于这种 unchecked exception，相对来说就会有点不一样。

这时就要用到 Thread 里面的 setUncaughtExceptionHandler(UncaughtExceptionHandler)，这个方法可以用来处理一些 unchecked exception。setUncaughtExceptionHandler()方法相当于一个事件注册的入口。在 Thread 类里面的源码如下：

```
    public void setUncaughtExceptionHandler(UncaughtExceptionHandler
paramUncaughtExceptionHandler)
    {
      checkAccess();
      this.uncaughtExceptionHandler = paramUncaughtExceptionHandler;
```

```
    }
```

而 UncaughtExceptionHandler 则是一个接口，它的声明如下：

```
    public static abstract interface UncaughtExceptionHandler
    {
        public abstract void uncaughtException(Thread paramThread, Throwable
paramThrowable);
    }
```

在异常发生的时候，我们传入的 UncaughtExceptionHandler 参数的 uncaughtException 方法会被调用。

综合前面的讨论，我们把要实现 handle unchecked exception 的方法的具体步骤总结如下：

- 定义一个类实现 UncaughtExceptionHandler 接口。在实现的方法里包含对异常处理的逻辑和步骤。
- 定义线程执行结构和逻辑。这一步和普通线程定义一样。
- 在创建和执行该子线程的方法中，在 thread.start() 语句前增加一个 thread.setUncaughtExceptionHandler 语句来实现处理逻辑的注册。

下面，我们就按照这个步骤来实现一个示例。

首先是实现 UncaughtExceptionHandler 接口部分：

```
    package demo.thread;
    import java.lang.Thread.UncaughtExceptionHandler;
    public class ExceptionHandlerThreadB implements UncaughtExceptionHandler {
        public void uncaughtException(Thread t, Throwable e) {
            System.out.printf("An exception has been captured\n");
            System.out.printf("Thread: %s\n", t.getId());
            System.out.printf("Exception: %s: %s\n", e.getClass().getName(),
e.getMessage());
            System.out.printf("Stack Trace: \n");
            e.printStackTrace(System.out);
            System.out.printf("Thread status: %s\n", t.getState());
        }
    }
```

这里我们添加的异常处理逻辑很简单，只是把线程的信息和异常信息都打印出来。

然后，我们定义线程的内容，这里，我们故意让该线程产生一个 unchecked exception：

```
    package demo.thread;
    public class ThreadB implements Runnable {
```

```
        public void run() {
            int number0 = Integer.parseInt("TTT");
        }
    }
```

从上面代码中我们可以看到，Integer.parseInt()里面的参数是错误的，肯定会抛出一个异常来。现在，我们再把创建线程和注册处理逻辑的部分补上来：

```
package demo.thread;
public class ThreadMain {
    public static void main(String[] args) {
        ThreadB task = new ThreadB();
        Thread thread = new Thread(task);
        thread.setUncaughtExceptionHandler( new ExceptionHandlerThreadB());
        thread.start();
    }
}
```

现在我们执行整个程序，会发现有如下的结果：

```
An exception has been captured
Thread: 8
Exception: java.lang.NumberFormatException: For input string: "TTT"
Stack Trace:
java.lang.NumberFormatException : For input string: "TTT"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at demo.thread.ThreadB.run( ThreadB.java:5)
    at java.lang.Thread.run(Unknown Source)
Thread status: RUNNABLE
```

这部分的输出正好就是我们前面实现 UncaughtExceptionHandler 接口的定义。

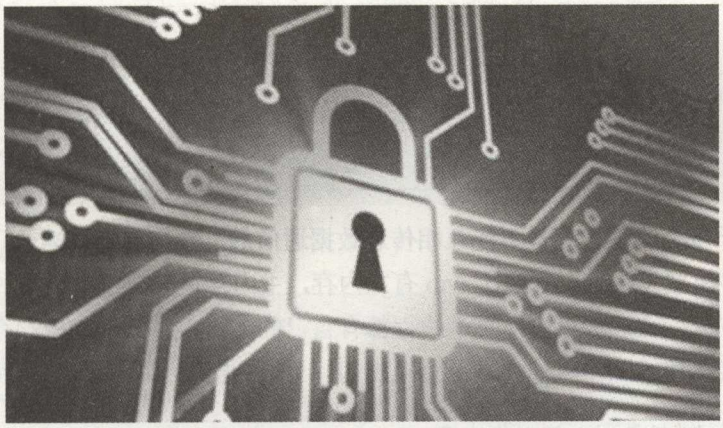
因此，对于 unchecked exception，我们也可以采用类似事件注册的机制做一定程度的处理。当然也可不处理放任自由，读者可以试试效果什么怎么样的，这里不实验了。

总之，Java thread 里面关于异常的部分比较奇特。你不能直接在一个线程里去抛出异常。一般在线程里碰到 checked exception，推荐的做法是采用 try/catch 块来处理。而对于 unchecked exception，比较合理的方式是注册一个实现 UncaughtExceptionHandler 接口的对象实例来处理。

第 3 章

Thread安全

安全两字很重要，不能忘记也不能丢。万一你把它忘了，程序就会出 Bug。



高并发、多线程开发的时候如何保证安全和程序的准确性？

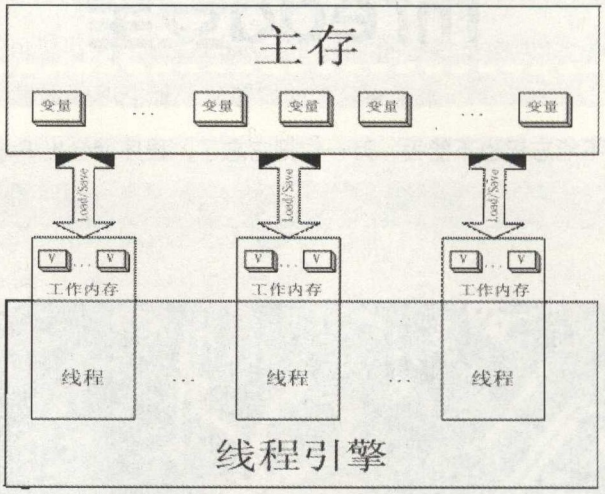
3.1 初识 Java 内存模型与多线程

想解决和认识安全性问题，先从原理上有个初浅的认识。

现代计算机，CPU 在计算的时候，并不总是从内存读取数据，它的数据读取顺序优先级是：寄存器→高速缓存→内存，线程计算的时候，原始的数据来自内存，在计算过程中，有些数据可能被频繁读取，这些数据被存储在寄存器和高速缓存中，当线程计算完后，这些缓存的数据在适当的时候应该写回内存，当多个线程同时读写某个内存数据时，由于涉及数据的可见性、操作的有序性，所以就会产生多线程并发问题。

Java 作为平台无关性语言，JLS（Java 语言规范）定义了一个统一的内存管理模型 JMM（Java Memory Model），JMM 屏蔽了底层平台内存管理细节，在多线程环境中必须解决可见性和有序性的问题。JMM 规定了 jvm 有主内存（Main Memory）和工作内存（Working Memory），主内存其实就是我们平时说的 Java 堆内存，存放程序中所有的类实例、静态数据等变量，是多个线程共享的，而工作内存存放的是该线程从主内存中拷贝过来的变量以及访问方法所取得的局部变量，

是每个线程私有的其他线程不能访问，每个线程对变量的操作都是以先从主内存将其拷贝到工作内存再对其进行操作的方式进行，多个线程之间不能直接互相传递数据通信，只能通过共享变量来进行。如下图所示。



问题一：由于多个线程之间是不能互相传递数据通信的，它们之间的沟通只能通过共享变量来进行。Java 内存模型（JMM）规定了 jvm 有主内存，主内存是多个线程共享的。当 new 一个对象的时候，也是被分配在主内存中，每个线程都有自己的工作内存，工作内存存储了主存的某些对象的副本，当然线程的工作内存大小是有限制的。当线程操作某个对象时，执行顺序如下：

（1）从主存复制变量到当前工作内存（read and load）
（2）执行代码，改变共享变量值 （use and assign）
（3）用工作内存数据刷新主存相关内容 （store and write）

所以单个线程与线程的工作内存之间就有了相互的隔离效果，专业术语上就称之为“可见性问题”。

问题二：线程在引用变量时不能直接从主内存中引用，如果线程工作内存中没有该变量，则会从主内存中拷贝一个副本到工作内存中，这个过程为 read-load，完成后线程会引用该副本。当同一线程再度引用该字段时，有可能重新从主存中获取变量副本（read-load-use），也有可能直接引用原来的副本（use），也就是说 read、load、use 顺序可以由 JVM 实现系统决定。这个时候线程与线程之间的操作的先后顺序，就会决定了你程序对主内存区最后的修改是不是正确的，专业术语上就称之为“时序性问题”。

3.2 什么是不安全

当多个线程同时操作一个数据结构的时候产生了相互修改和串行的情况，没有保证数据的一

致性，我们通常称这种设计的代码为“线程不安全的”。

有这么一个场景，假设 5 个用户，都来给一个数字加 1 的工作，那么最后应该是得到加 5 的结果；看一下下面的事例。

单个用户干活类：Count 。

```
public class Count {
        public int num = 0;
        public void add() {
              try {
                    Thread. sleep(51);//模仿用户干活
              } catch (InterruptedException e) {
              }
              num += 1;
              System. out.println(Thread.currentThread().getName() + "-" + num);
        }
}
```

用户类，干 Count 的活。

```
package demo.thread;
public class ThreadA extends Thread {
        private Count count ;
        public ThreadA(Count count) {
              this.count =count;
        }
        public void run() {
              count.add();
        }
}
```

5 个人干完活：最后的值。

```
package demo.thread;
public class ThreadMain {
        public static void main(String[] args) {
              Count count = new Count();
              for(int i=0;i<5;i++) {
                    ThreadA task = new ThreadA(count);
                    task.start();
              }
              try {
```

```
                Thread. sleep(1001);//等5个人干完活
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System. out.println("5个人干完活:最后的值4" +count .num );
        }
}
```

运行结果如下（由于多线程，有不安全问题，其实每次运行下面的结果都是不一样的）：

```
Thread-1-1
Thread-0-1
Thread-3-3
Thread-4-2
Thread-2-3
5个人干完活:最后的值: 3
```

可见不是我们想要的结果，这就是典型的线程不安全问题。在我们实际工作中，特别是 Web 项目 Service 和 servlet 一般都是单例共享变量的情况下极其容易出现，多个用户之间的数据串掉了，从而导致最终数据库里面所需要统计的数据不对。

3.3 什么是安全

我们把上一节的例子单个用户干活类 Count 做如下修改，添加 **synchronized** 关键字。

```
package demo.thread;
public class Count {
        public int num = 0;
        public synchronized void add() {
            try {
                Thread. sleep(51);//模仿用户干活
            } catch (InterruptedException e) {
            }
            num += 1;
            System. out.println(Thread.currentThread().getName() + "-" + num);
        }
}
```

运行结果如下：

```
Thread-0-1
Thread-4-2
Thread-3-3
Thread-2-4
Thread-1-5
5个人干完活:最后的值5
```

而这次，每次干活都是一样的结果，这就叫线程安全，就是不管多少个用户过来，都保证我们的数据的高度一致性和准确性就叫线程安全的；这里我们引用了 synchronized 的同步锁的机制，这个后面会讲到，它保证了我们的线程安全性。

什么是线程安全性呢？是不是一定要加锁才是线程安全性的呢？个人感觉只要你代码里面没有变量互串，线程之间互不影响，例如 server 的设计方法，就是线程安全的，例如上面 5 个人干了同一件事情，如果让 5 个人干 5 件不一样的事情，或者 1 个人干 5 件事情，那也是安全的。而不安全在 Java 工作中主要针对单例模式的应用而言的，怎么保证一件事情被一群人干完，又快又正确。

想实现线程安全大致有三种方法。

- 多实例，也就是不用单例模式了。
- 使用 java.util.concurrent 下面的类库。
- 使用锁机制 synchronized、lock 方式。

3.4 隐式锁，又称线程同步 synchronized

synchronized 是 Java 语言的关键字，当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码。其实简单一点理解，我感觉就是要解决我们前面说的多线程并发时候的“时序性问题”，即访问要有一个顺序，讲究先来后到，就看谁先能拿到这个锁对象。

synchronized 的用法，修饰地方有只有两个：

一是在方法声明时使用，放在范围操作符（public 等）之后，返回类型声明（void 等）之前的方法名上面，代码如下：

```
public synchronized void synMethod() {
        //方法体
}
```

二是修饰在代码块上面的，对某一代码块使用 synchronized(Object)，指定加锁对象：

```
public int synMethod(int a1){
```

```
        synchronized(this) {
        //一次只能有一个线程进入
        }
}
```

其中 synchronized(Object)中的 Object 可以是任意对象，可以是参数本身，可以是当前对象 this，也可以是指定的对象；作者不建议使用方法参数，也不建议使用 this 对象，而是另外指定一个小的对象值。很多资料和文档都称它为同步锁，而我更愿意叫它隐式锁。因为尽管它修饰的地方不尽相同，但是最终它的锁都在一个对象上面；修饰在方法体上的 synchronized 默认锁的对象就是当前对象本身；等同于 synchronized (this) {}修饰代码块的用法；或者是你指定的 Object 对象作为锁，因为和它持有相同对象锁的地方将产生互斥性，而不是只有当前所指的代码块或者方法体，另外一个就是相对显示锁不需要加锁和解锁的操作，所以称它为隐式的。

有一些隐式规则如下：

（1）当两个并发线程访问同一个对象 object 中的这个 synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

（2）然而，当一个线程访问 object 的一个 synchronized(this)同步代码块时，另一个线程仍然可以访问该 object 中的非 synchronized(this)同步代码块。

（3）尤其关键的是，当一个线程访问 object 的一个 synchronized(this)同步代码块时，其他线程对 object 中所有其他 synchronized(this)同步代码块的访问将被阻塞。

（4）第三个规则同样适用其他同步代码块。也就是说，当一个线程访问 object 的一个 synchronized(this)同步代码块时，它就获得了这个 object 的对象锁。结果，其他线程对该 object 对象所有同步代码部分的访问都被暂时阻塞。

（5）以上规则对其他对象锁同样适用。

下面举例加以说明：

看一个正确的例子：尽管 synchronized 的写法不一样，但是这两个方法对于多线程来说是线程安全的。

```
package demo.thread;
public class Count {
      public int num = 0;
      public synchronized void methodA() {
            try {
                  Thread. sleep(5l);// 模仿用户干活
            } catch (InterruptedException e) {
            }
             num += 1;
```

```
            System. out.println(Thread.currentThread().getName() + "-" + num);
        }
        public void methodB() {
            synchronized (this ) {
                try {
                    Thread. sleep(5l);// 模仿用户干活
                } catch (InterruptedException e) {
                }
                num += 1;
                System. out.println(Thread.currentThread().getName()
+ "-" + num );
            }
        }
    }
```

看一个错误的例子：

```
package demo.thread;
public class Count {
    private byte [] lock = new byte[1];
    public int num = 0;
    public synchronized void methodA() {
        try {
            Thread. sleep(5l);// 模仿用户干活
        } catch (InterruptedException e) {
        }
        num += 1;
        System. out.println(Thread.currentThread().getName() + "-" + num);
    }
    public void methodB() {
        synchronized (lock ) {//注意这个锁的对象不一样
            try {
                Thread. sleep(5l);// 模仿用户干活
            } catch (InterruptedException e) {
            }
            num += 1;
            System. out.println(Thread.currentThread().getName()
+ "-" + num );
        }
```

```
        }
    }
```

为什么说这两个方法对于多线程来说是线程不安全的呢？因为它锁定的对象不一样，所以作者不建议用参数作为锁的对象，那样子，你的同步锁只会对这个方法有用，而失去了synchronized 锁定对象的意义了。

为适用高并发对性能以及快速响应的要求，synchronized 不同的写法程序响应的快慢和对CPU 等资源高并发的利用程度又不一样；性能和执行效率的优劣程度从差到优有如下排序：

同步方法体

```
public synchronized void synMethod() {
        //方法体
}
```

小于

```
public int synMethod(int a1){
        synchronized(this) {
        //一次只能有一个线程进入
        }
}
```

因为一个是进入方法体内加锁的，一个是排队在方法体外，这样即使获得了锁，进入方法体还得分配资源需要一定的时间。

而这种方法块锁又小于下面的

```
private byte [] lock = new byte [1];
public void methodB() {
        synchronized (lock ) { //注意这个锁的对象不一样
        // TODO
    }
}
```

因为锁的对象不一样，锁是对象，加锁和释放锁都需要此对象的资源，那肯定对象越小越好啊，所以造一个一个字节的 byte 对象最小；所以工作中这种写法最常见。

3.5 显示锁 Lock 和 ReentrantLock

Lock 是一个接口提供了无条件的、可轮询的、定时的、可中断的锁获取操作，所有加锁和解锁的方法都是显式的。包路径是：java.util.concurrent.locks.Lock。核心方法是 lock()、unlock()、tryLock()，实现类有 ReentrantLock、ReentrantReadWriteLock.ReadLock、 ReentrantReadWriteLock.WriteLock。

看一下 Lock 接口有如下方法：

```
public abstract interface Lock
{
    public abstract void lock();
    public abstract void lockInterruptibly()  throws InterruptedException;
    public abstract boolean tryLock();
    public abstract boolean tryLock(long paramLong , TimeUnit paramTimeUnit)
    throws InterruptedException;
    public abstract void unlock();
    public abstract Condition newCondition();
}
```

对应的解说如下：

void lock(); 获取锁。如果锁不可用，出于线程调度目的，将禁用当前线程，并且在获得锁之前，该线程将一直处于休眠状态。

void lockInterruptibly() throws InterruptedException; 如果当前线程未被中断，则获取锁。如果锁可用，则获取锁，并立即返回。如果锁不可用，出于线程调度目的，将禁用当前线程，并且在发生以下两种情况之一以前，该线程将一直处于休眠状态：锁由当前线程获得；或者其他某个线程中断当前线程，并且支持对锁获取的中断。如果当前线程：在进入此方法时已经设置了该线程的中断状态；或者在获取锁时被中断 ，并且支持对锁获取的中断，则将抛出 InterruptedException ，并清除当前线程的已中断状态。

boolean tryLock(); 仅在调用时锁为空闲状态才获取该锁。如果锁可用，则获取锁，并立即返回值 true 。如果锁不可用，则此方法将立即返回值 false 。通常对于那些不是必须获取锁的操作可能有用。

boolean tryLock(long ParamLong, TimeUnit ParamTimeUnit) throws InterruptedException; 如果锁在给定的等待时间内空闲，并且当前线程未被中断，则获取锁。如果锁可用，则此方法将立即返回值 true 。如果锁不可用，出于线程调度目的，将禁用当前线程，并且在发生以下三种情况的任一种之前，该线程将一直处于休眠状态：

void unlock(); 释放锁。对应于 lock()、tryLock()、tryLock(xx)、lockInterruptibly()等操作，如果成功的话应该对应着一个 unlock()，这样可以避免死锁或者资源浪费。

new Condition(); 返回用来与此 Lock 实例一起使用的 Condition 实例。

ReentrantLock 是 Lock 的实现类，是一个互斥的同步器，它具有扩展的能力。在竞争条件下，ReentrantLock 的实现要比现在的 synchronized 实现更具有可伸缩性。（有可能在 JVM 的将来版本中改进 synchronized 的竞争性能）这意味着当许多线程都竞争相同锁定时，使用 ReentrantLock 的吞吐量通常要比 synchronized 好。换句话说，当许多线程试图访问 ReentrantLock 保护的共享资源时，JVM 将花费较少的时间来调度线程，而用更多个时间执行线程。虽然 ReentrantLock 类有许多优点，但是与同步相比，它有一个主要缺点：它可能忘记释放锁定。ReentrantLock 是在工作中对方法块加锁使用频率最高的。

使用方法如下：

```
class X {
      private final ReentrantLock lock = new ReentrantLock();
      // ...
      public void m() {
            lock.lock(); // 获得锁
            try {
                  // ... 方法体
            } finally {
                  lock.unlock();//解锁
            }
      }
}
```

Lock 与 synchronized 的比较：

（1）Lock 使用起来比较灵活，但是必须有释放锁的动作配合。

（2）Lock 必须手动释放和开启锁，而 synchronized 不需要手动释放和开启锁。

（3）Lock 只适用与代码块锁，而 synchronized 对象之间是互斥关系。

请注意以下两种方式的区别：

第一种方式：两个方法之间的锁是独立的。代码如下：

```
public class ReentrantLockDemo {
      public static void main(String[] args) {
            final Count ct = new Count();
            for (int i = 0; i < 2; i++) {
                  new Thread() {
                        @Override
                        public void run() {
                              ct.get();
                        }
```

```
                }.start();
            }

            for (int i = 0; i < 2; i++) {
                new Thread() {
                    @Override
                    public void run() {
                        ct.put();
                    }
                }.start();
            }
        }
    }
    class Count {
        public void get() {
            final ReentrantLock lock = new ReentrantLock();
            try {
                lock.lock(); // 加锁
                System. out.println(Thread.currentThread().getName()
+ "get begin");
                Thread. sleep(1000L);// 模仿干活
                System. out.println(Thread.currentThread().getName()
+ "get end");
                lock.unlock(); // 解锁
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        public void put() {
            final ReentrantLock lock = new ReentrantLock();
            try {
                lock.lock(); // 加锁
                System. out.println(Thread.currentThread().getName()
+ "put begin");
                Thread. sleep(1000L);// 模仿干活
                System. out.println(Thread.currentThread().getName()
+ "put end");
                lock.unlock(); // 解锁
            } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }
            }
        }
```

运行结果如下（每次运行结果都是不一样的，仔细体会一下）：

```
Thread-0get begin
Thread-1get begin
Thread-2put begin
Thread-3put begin
Thread-0get end
Thread-2put end
Thread-3put end
Thread-1get end
```

第二种方式，两个方法之间使用相同的锁。

ReentrantLockDemo 类的内容不变，将 Count 中的 ReentrantLock 改成全局变量，如下所示：

```
class Count {
        final ReentrantLock lock = new ReentrantLock();
        public void get() {
                try {
                        lock.lock(); // 加锁
                        System. out.println(Thread.currentThread().getName()
+ "get begin");
                        Thread. sleep(1000L);// 模仿干活
                        System. out.println(Thread.currentThread().getName()
+ "get end");
                        lock.unlock(); // 解锁
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
        public void put() {
                try {
                        lock.lock(); // 加锁
                        System. out.println(Thread.currentThread().getName()
+ "put begin");
                        Thread. sleep(1000L);// 模仿干活
                        System. out.println(Thread.currentThread().getName()
```

```
+ "put end");
                        lock.unlock(); // 解锁
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
    }
```

运行结果如下（每次运行结果是一样的，仔细体会一下）：

```
Thread-0get begin
Thread-0get end
Thread-1get begin
Thread-1get end
Thread-2put begin
Thread-2put end
Thread-3put begin
Thread-3put end
```

3.6 显示锁 ReadWriteLock 和 ReentrantReadWriteLock

ReadWriteLock 也是一个接口，提供了 readLock 和 writeLock 两种锁的操作机制，也就是一个资源能够被多个读线程访问，或者被一个写线程访问，但是不能同时存在读写线程。也就是说读写锁使用的场合是一个共享资源被大量读取操作，而只有少量的写操作。包路径是 java.util.concurrent.locks.ReadWriteLock。

我们来看一下 ReadWriteLock 的源码：

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

从源码上面我们可以看出来 ReadWriteLock 并不是 Lock 的子接口，只不过 ReadWriteLock 借助 Lock 来实现读写两个锁并存、互斥的操作机制。在 ReadWriteLock 中每次读取共享数据就需要读取锁，当需要修改共享数据时就需要写入锁。看起来好像是两个锁，但是并非如此。

ReentrantReadWriteLock 是 ReadWriteLock 在 java.util 里面的唯一的实现类，主要应用场景是

当有很多线程都从某个数据结构中读取数据，而很少有线程对其进行修改时。在这种情况下，允许读取器线程共享访问是合适的，写入器线程依然必须是互斥访问的。

ReentrantReadWriteLock 的实现里面有以下几个特性：

（1）公平性：非公平锁（默认）。这个和独占锁的非公平性一样，由于读线程之间没有锁竞争，所以读操作没有公平性和非公平性，写操作时，由于写操作可能立即获取到锁，所以会推迟一个或多个读操作或者写操作。因此，非公平锁的吞吐量要高于公平锁。公平锁利用 AQS 的 CLH 队列，释放当前保持的锁（读锁或者写锁）时，优先为等待时间最长的那个写线程分配写入锁，当前前提是写线程的等待时间要比所有读线程的等待时间要长。同样一个线程持有写入锁或者有一个写线程已经在等待了，那么试图获取公平锁的（非重入）所有线程（包括读写线程）都将被阻塞，直到最先的写线程释放锁。如果读线程的等待时间比写线程的等待时间还要长，那么一旦上一个写线程释放锁，这一组读线程将获取锁。

（2）重入性：读写锁允许读线程和写线程按照请求锁的顺序重新获取读取锁或者写入锁。当然了，只有写线程释放了锁，读线程才能获取重入锁。写线程获取写入锁后可以再次获取读取锁，但是读线程获取读取锁后却不能获取写入锁。另外读写锁最多支持 65535 个递归写入锁和 65535 个递归读取锁。

（3）锁降级：写线程获取写入锁后可以获取读取锁，然后释放写入锁，这样就从写入锁变成了读取锁，从而实现锁降级的特性。

（4）锁升级：读取锁是不能直接升级为写入锁的。因为获取一个写入锁需要释放所有读取锁，所以如果有两个读取锁视图获取写入锁，且都不释放读取锁时就会发生死锁。

（5）锁获取中断：读取锁和写入锁都支持获取锁期间被中断。这个和独占锁一致。

（6）条件变量：写入锁提供了条件变量（Condition）的支持，这个和独占锁一致，但是读取锁却不允许获取条件变量，否则会得到一个 UnsupportedOperationException 异常。

（7）重入数：读取锁和写入锁的数量最大分别只能是 65535。

概括起来其实就是读写锁的机制：

（1）读-读不互斥，比如当前有 10 个线程去读（没有线程去写），这个 10 个线程可以并发读，而不会堵塞。

（2）读-写互斥，当前有写的线程的时候，那么读取线程就会堵塞，反过来，有读的线程在使用的时候，写的线程也会堵塞，就看谁先拿到锁了。

（3）写-写互斥，写线程都是互斥的，如果有两个线程去写，A 线程先拿到锁就先写，B 线程就堵塞直到 A 线程释放锁。

使用读/写锁的方法和步骤：

```
// 创建一个 ReentrantReadWriteLock 对象
private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
//抽取读锁和写锁
```

```
    private Lock readLock = rwl.readLock(); //得到一个可被多个读操作共用的读锁，但它会排斥所有写操作
    private Lock writeLock = rwl.writeLock(); //得到一个写锁，它会排斥所有其他的读操作和写操作
    //对所有访问者加读锁
    public double getTotalBalance(){
            readLock.lock();
            try{...};
            finally{readLock.unlock();}
    }
    //对所有修改者加写锁
    public void transfer(){
            writeLock.lock();
            try{...};
            finally{writeLock.unlock();}
    }
```

实例体会：

第一种情况，先体验一下 ReadLock 和 WriteLock 单独使用的情况。

```
public static void main(String[] args) {
            final Count ct = new Count();
            for (int i = 0; i < 2; i++) {
                    new Thread() {
                            public void run() {
                                    ct.get();
                            }
                    }.start();
            }
            for (int i = 0; i < 2; i++) {
                    new Thread() {
                            public void run() {
                                    ct.put();
                            }
                    }.start();
            }
    }
class Count {
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    public void get() {
```

```
                rwl.readLock().lock(); // 上读锁，其他线程只能读不能写，具有高并发性
                try {
                        System. out.println(Thread.currentThread().getName()   + "
read start.");
                        Thread. sleep(1000L);// 模拟干活
                        System. out.println(Thread.currentThread().getName()
+ "read end.");
                } catch (InterruptedException e) {
                        e.printStackTrace();
                } finally {
                        rwl.readLock().unlock(); // 释放读锁，最好放在 finnaly 里面
                }
        }
        public void put() {
                rwl.writeLock().lock(); // 上写锁，具有阻塞性
                try {
                        System. out.println(Thread.currentThread().getName()   + "
write start.");
                        Thread. sleep(1000L);// 模拟干活
                        System. out.println(Thread.currentThread().getName()
+ "write end.");
                } catch (InterruptedException e) {
                        e.printStackTrace();
                } finally {
                        rwl.writeLock().unlock(); // 释放写锁，最好放在 finnaly 里面
                }
        }
    }
```

运行结果如下（从结果上面可以看的出来，读的时候是并发的，写的时候是有顺序的带阻塞机制的）：

```
Thread-0 read start.
Thread-1 read start.
Thread-1 read end.
Thread-0 read end.
Thread-3 write start.
Thread-3 write end.
Thread-2 write start.
Thread-2 write end.
```

第二种情况，我们体会一个 ReadLock 和 WriteLock 的复杂使用情况，模拟一个有读写数据的场景，仔细体会一下。

```
    private final Map<String, Object> map = new HashMap<String, Object>(); // 假设这里面存了数据缓存
    private final ReentrantReadWriteLock rwlock = new ReentrantReadWriteLock();
  public Object readWrite(String id) {
            Object value = null;
            rwlock.readLock().lock(); // 首先开启读锁，从缓存中去取
            try {
                  value = map.get(id);
                  if (value == null) { // 如果缓存中没有释放读锁，上写锁
                        rwlock.readLock().unlock();
                        rwlock.writeLock().lock();
                        try {
                              if (value == null) {
                                    value = "aaa"; // 此时可以去数据库中查找，这里简单的模拟一下
                              }
                        } finally {
                              rwlock.writeLock().unlock(); // 释放写锁
                        }
                        rwlock.readLock().lock(); // 然后再上读锁
                  }
            } finally {
                  rwlock.readLock().unlock(); // 最后释放读锁
            }
            return value;
  }
```

ReentrantReadWriteLock 与 ReentrantLock 的比较：

（1）相同点：其实都是一种显示锁，手动加锁和解锁，都很适合高并发场景。

（2）不同点：ReentrantReadWriteLock 是对 ReentrantLock 的复杂扩展，能适合更加复杂的业务场景，ReentrantReadWriteLock 可以实现一个方法中读写分离的锁的机制。而 ReentrantLock 加锁解锁只有一种机制。

3.7 显示锁 StampedLock

在理解 StampedLock 之前，不知道大家之前接触过悲观锁和乐观锁没有，我们在这里先做一个简单说明：

- 悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。
- 读取悲观锁：在读取之前一定要判断一下，数据又没有正在被更改。
- 乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。
- 读取乐观锁：在读取之前就不需要来判断数据的一致性，我只管读我自己的就可以了。

StampedLock 是基于能力的锁，可以很好地实现悲观锁和乐观锁的逻辑。它使用三种模式来控制读/写访问。StampedLock 的状态包含了版本和模式。锁获取方法根据锁的状态返回一个表示和控制访问的标志（stamp），“try”版本的这些方法可能返回一个特殊的值 0 来表示获取失败。锁释放和它的变换方法要求一个标志作为参数，如果它们不符合锁的状态就失败。这三种模式分别是：

- 写：方法 writeLock 可能阻塞等待独占访问，返回一个标志，可用在方法 unlockWrite 以释放锁。也提供了无时间和带时间版本的 tryWriteLock 方法。当锁以写模式持有时，没有读锁可以获取，所有乐观性读确认将失败。
- 读 方法 readLock 可能为非独占访问而阻塞等待，返回一个标志用于方法 unlockRead 以释放锁。也提供了无时间和带时间版本的 tryWriteLock 方法。
- 乐观读：只有在锁当前没有以写模式持有时，方法 tryOptimisticRead 返回一个非 0 标志。如果锁自给定标志以来没有以写模式持有，方法 validate 返回 true 。这种模式可以认为是一种极弱版本的读锁，可以在任意时间被写者打破。在短的只读代码段使用乐观模式常常可以减少竞争并提升吞吐量。然而，它的使用天生是脆弱的。乐观读片段 section 应该只读字段并持有到本地变量，用于以后使用，在确认以后。乐观读模式里的字段读取可能很不一致，所以惯例只用于当你对数据表示足够熟悉，可以检查一致性和/或重复调用 validate() 方法。例如，这些步骤典型地在第一次读取对象或数组引用，然后访问其中字段、元素或方法时要求。

我们来看一下 StampedLock 的部分源码：

```
public class StampedLock implements Serializable {
private volatile transient long state = 256L;//状态标志(stamp)
transient ReadLockView readLockView;//ReadLock 和 WriteLock 都是 Lock 的实现
transient WriteLockView writeLockView;//
transient ReadWriteLockView readWriteLockView; //实现 ReadWriteLock 接口
        final class ReadWriteLockView implements ReadWriteLock {
                public Lock readLock() {
```

```
                return StampedLock.this.asReadLock();
            }
            public Lock writeLock() {
                return StampedLock.this.asWriteLock();
            }
        }
        final class ReadLockView implements Lock {
            public void lock() {
                StampedLock. this.readLock();
            }
            public void unlock() {
                StampedLock. this.unstampedUnlockRead();
            }
            //……
        }
        final class WriteLockView implements Lock {
            public void lock() {
                StampedLock. this.writeLock();
            }
            //……
        }

// ……
}
```

由此可以看得出来 StampedLock 也是利用 Lock 机制，再加上 stamp 作为锁的标志状态，实现了锁与锁之间的悲观和乐观。

这个类也支持方法以有条件地提供三种模式之间的转换。例如，方法 tryConvertToWriteLock 尝试“升级”模式，返回一个有效的写标志。[1]表示已经是写模式，[2]表示在读模式且没有其他读者，[3]表示在乐观模式且锁可得。这些方法的形式是设计用于帮助减少一些代码膨胀，否则将出现在基于重试的设计。

StampedLock 设计用作开发线程安全组件的内部工具。它们的使用依赖于对数据、对象和它们所保护方法的内在属性的知识。它们不是可重入的，所以锁保护块不应该调用其他可能尝试再次获取锁（虽然你可以传递标志给其他方法，然后使用或转换它）的未知方法。使用读锁模式依赖于关联的代码片段是无边际效应的（也就是无副作用）。无效的乐观读片段不能调用那些不知道忍受潜在不一致的方法。标志使用有限的表示，且不是密码加密安全的（例如，一个有效的标志是可猜测的）。在（不早于）一年的持续操作后，标志的值可能会循环。持有标志而不使用或确认，在大于这个周期可能将不能正确确认。StampedLock 是可序列化的，但总是反序列化为初

始未锁定状态，所以对于远程锁定没有帮助。

StampedLock 的调度策略不是一惯地倾向于选择读而不是写，或相反。所有“try”方法都是尽最大努力的，不必向任何调度或公平策略确认。任何用于获取的“try”方法或不带任何关于锁状态的信息的转换锁模式的方法返回 0 ；后续的调用可能成功。

因为它支持协调跨多种锁模式使用，这个类不直接使用 Lock 或 ReadWriteLock 接口。然而，在要求这样一组关联功能的应用里，StampedLock 可以看作 asReadLock()、asWriteLock()或 asReadWriteLock()。

下面是 Java 的 doc 中提供的一个例子：

```
class Point {
    private double x , y ;
    private final StampedLock sl = new StampedLock();
    void move(double deltaX, double deltaY) { // an exclusively locked method
        long stamp = sl .writeLock();
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }
    // 下面看看乐观读锁案例
    double distanceFromOrigin() { // A read-only method
        long stamp = sl .tryOptimisticRead(); // 获得一个乐观读锁
        double currentX = x , currentY = y; // 将两个字段读入本地局部变量
        if (!sl .validate(stamp)) { // 检查发出乐观读锁后同时是否有其他写锁发生?
            stamp = sl.readLock(); // 如果没有，我们再次获得一个读悲观锁
            try {
                currentX = x; // 将两个字段读入本地局部变量
                currentY = y; // 将两个字段读入本地局部变量
            } finally {
                sl.unlockRead(stamp);
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY);
    }
    // 下面是悲观读锁案例
    void moveIfAtOrigin(double newX, double newY) { // upgrade
        // Could instead start with optimistic, not read mode
```

```
                long stamp = sl .readLock();
                try {
                      while (x == 0.0 && y == 0.0) { // 循环，检查当前状态是否符合
                            long ws = sl .tryConvertToWriteLock(stamp); // 将读锁转
为写锁
                            if (ws != 0L) { // 这是确认转为写锁是否成功
                                  stamp = ws; // 如果成功 替换票据
                                  x = newX; // 进行状态改变
                                  y = newY; // 进行状态改变
                                  break;
                            } else { // 如果不能成功转换为写锁
                                  sl.unlockRead(stamp); // 我们显式释放读锁
                                  stamp = sl.writeLock(); // 显式直接进行写锁 然后再通过
循环再试
                            }
                      }
                } finally {
                      sl.unlock(stamp); // 释放读锁或写锁
                }
          }
    }
```

StampedLock 是 JDK 1.8 之后新推出的一个 API，可大幅度提高程序的读取锁的吞吐量。在大多数都是读取、很少写入的情况下，乐观读锁模式可以极大提供吞吐量，也可以减少这种情况下写饥饿的现象（由于读者一般不加读锁，写可以马上获取到锁）。

最后 Synchronized、ReentrantLock、ReentrantReadWriteLock、StampedLock 简单对比如下：

（1）Synchronized 是在 JVM 层面上实现的，可以通过一些监控工具监控 synchronized 的锁定，当代码执行时出现异常，JVM 会自动释放锁定。当只有少量竞争者的时候，synchronized 是一个很好的通用的锁实现。Synchronized 的锁是针对一个对象的。

（2）ReentrantLock、ReentrantReadWriteLock,、StampedLock 都是代码块层面的锁定，要保证锁定一定会被释放，就必须将 unLock()放到 finally{}中。

（3）ReentrantLock 是一个很好的通用的锁实现，使用于比较简单的加锁、解锁的业务逻辑，如果实现复杂的锁机制，当线程增长能够预估时也是可以的。

（4）ReentrantReadWriteLock 对 Lock 又进行了扩展，引入了 read 和 write 阻塞和并发机制，相对于 ReentrantLock，它可以实现更复杂的锁机制，且并发性也更高些。

（5）StampedLock 又在 Lock 的基础上，实现了可以满足乐观锁和悲观锁等一些在读线程越来越多的业务场景，对吞吐量有巨大的改进，但并不是说要替代之前的 Lock，毕竟它还是有些应用场景的。

（6）StampedLock 有一个复杂的 API 相对于前面两种 Lock 锁，对于加锁操作，很容易误用其他方法，如果理解不深入也更容易出现死锁和不必要的麻烦。

（7）推荐如果不是业务非得需要，建议使用 ReentrantLock 和 ReentrantReadWriteLock 即可满足大部分业务场景的需要。

3.8 什么是死锁

在两段不同的逻辑都在等待对方的锁释放才能继续往下工作时，这个时候就会产生死锁，表面现象就是程序再也执行不下去了。

请看如下事例：新建一个干活类的两个不同方法在等待锁。

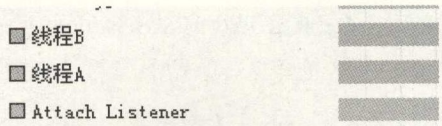
```
public class Count {
        private byte [] lock1 = new byte[1];
        private byte [] lock2 = new byte[1];
        public int num = 0;
        public void add() {
              synchronized (lock1 ) {// 注意这个锁的对象不一样
                    try {
                          Thread. sleep(1001);// 模仿用户干活
                    } catch (InterruptedException e) {
                    }
                    synchronized (lock2 ) {// 产生死锁等待 lock2对象释放锁
                          num += 1;
                    }
                    System. out.println(Thread.currentThread().getName()
+ "-" + num );
              }
        }
        public void lockMethod() {
              synchronized (lock2 ) {// 注意这个锁的对象不一样
                    try {
                          Thread. sleep(1001);// 模仿用户干活
                    } catch (InterruptedException e) {
                    }
                    synchronized (lock1 ) {// 产生死锁等待 lock1对象释放锁
                          num += 1;
                    }
```

```
                    System. out.println(Thread.currentThread().getName()
+ "-" + num );
                }
            }
        }
```

做两个线程一个调用 add 的方法，一个调用 lockMethod 方法。

```
public class ThreadB extends Thread {
        private Count count ;
        public ThreadB(Count count) {
                this.count =count;
        }
        public void run() {
                count.lockMethod();
        }
}
public class ThreadA extends Thread {
        private Count count ;
        public ThreadA(Count count) {
                this.count =count;
        }
        public void run() {
                count.add();
        }
}
public static void main(String[] args) {
            Count count = new Count();
            ThreadA task = new ThreadA(count);
            task.setName( "线程A");
            task.start();
            ThreadB taskB = new ThreadB(count);
            taskB.setName( "线程B");
            taskB.start();
}
```

通过工具来监控线程 A 和线程 B 永远在这里：



通过执行上面的 main 方法你会发现，程序死在那里，永远不会执行下去了，而实际工作中应该注意的是方法之间嵌套调用上不能产生死锁。

3.9 Java 关键字 volatile 修饰变量

Java 关键字 volatile 修饰变量，从表面意思上是说这个变量是易变的，不稳定的，事实上，确实如此，这个关键字的作用就是告诉编译器，凡是被该关键字声明的变量都是易变的、不稳定的。所以不要试图对该变量使用缓存等优化机制，而应当每次都从它的内存地址中去读取值。使用 volatile 标记的变量在读取或写入时不需要使用锁，这将减少产生死锁的概率，使代码保持简洁。

请注意，这里只是说每次读取 volatile 的变量时都要从它的内存地址中读取，并没有说每次修改完 volatile 的变量后都要立刻将它的值写回内存。也就是说 volatile 只提供了内存可见性，而没有提供原子性。所以说如果用这个关键字做高并发的安全机制的话是不可靠的。

volatile 的用法如下：

```
public volatile static int count = 0;
```

在声明变量的时候带上 volatile 关键字即可。什么时候使用 volatile 关键字？当我们知道了 volatile 的作用，我们也就知道了它应该用在哪些地方。很显然，最好是那种只有一个线程修改变量，多个线程读取该变量的地方。也就是对内存可见性要求高，而对原子性要求低的地方。

volatile 与加锁机制的主要区别是：加锁机制既可以确保可见性又可以确保原子性，而 volatile 变量只有确保可见性。

3.10 原子操作：atomic

atomic 是不会阻塞线程（或者说只是在硬件级别上阻塞了），线程安全的加强版的 volatile 原子操作。java.util.concurrent.atomic 包里，多了一批原子处理类，主要用于在高并发环境下的高效程序处理。这些处理类主要有以下几种：

- 基本类：AtomicInteger、AtomicLong、AtomicBoolean。
- 引用类型：AtomicReference、AtomicReference 的 ABA 实例、AtomicStampedRerence、AtomicMarkableReference。

- 数组类型：AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray。
- 属性原子修改器（Updater）：AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater。

我们以其中一个为使用例子 AtomicInteger。其余的方法和使用都是大同小异的，相关的类会介绍它们之间的区别在哪里，在使用中需要注意的地方即可。而 AtomicInteger 这个类，特别适用于高并发访问。

AtomicInteger 的主要方法有：

```
//获取当前的值
public final int get()
//取当前的值，并设置新的值
public final int getAndSet(int newValue)
//获取当前的值，并自增
public final int getAndIncrement()
//获取当前的值，并自减
public final int getAndDecrement()
//获取当前的值，并加上预期的值
public final int getAndAdd(int delta)
```

使用方法如下：

```
public static void main(String[] args) {
        AtomicInteger ai = new AtomicInteger(0);
        System. out.println(ai.get());
        System. out.println(ai.getAndSet(5));
        System. out.println(ai.getAndIncrement());
        System. out.println(ai.getAndDecrement());
        System. out.println(ai.getAndAdd(10));
        System. out.println(ai.get());
}
```

输出的结果为：

```
0
0
5
6
5
15
```

原子操作 atomic 的实现原理，是利用 CPU 的比较并交换（即 CAS:Compare and Swap）和非

阻塞算法（nonblocking algorithms）；如果查看 AtomicInteger 的源码的话会发现有些是通过调用 JNI 的代码实现的。JNI（Java Native Interface）为 JAVA 本地调用，允许 Java 调用其他语言。而 compareAndSwapInt 就是借助 C 来调用 CPU 底层指令实现的。基于 JAVA CAS 原理深度分析这里就不多说了，我们 Java 程序员了解一些就行了。

3.11 单利模式的写法

利用前面所学习的知识，我们根据锁的原理来比较一下下面四种单例模式实例。

第一种：线程不安全的，不正确的写法：

```
class Singleton {
      private static Singleton instance;
      private Singleton() {
      }
      public static Singleton getInstance() {
            if (instance == null) {
                  instance = new Singleton();
            }
            return instance ;
      }
}
```

第二种：线程安全性，但是高并发性能不是很高的写法：

```
class Singleton {
      private static Singleton instance;
      private Singleton() {
      }
      public static synchronized Singleton getInstance() {
            if (instance == null) {
                  instance = new Singleton();
            }
            return instance ;
      }
}
```

第三种：线程安全，性能又高的，这种写法最为常见：

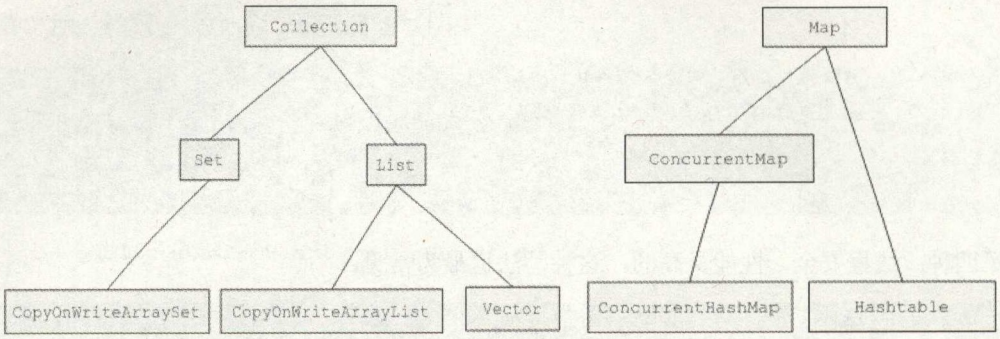
```
class Singleton {
      private static Singleton instance ;
      private static byte[] lock = new byte [0];
      private Singleton() {
      }
      public static Singleton getInstance() {
         if (instance == null) {
             synchronized (lock ) {
                   if (instance == null) {
                         instance = new Singleton ();
                   }
             }
         }
               return instance ;
      }
}
```

第四种：线程安全，性能又高的，这种写法也最为常见：

```
class Singleton {
       private static Singfleton instance;
       private static ReentrantLock lock = new ReentrantLock();
       private Singleton() {
       }
       public static Singleton getInstance() {
              if (instance == null) {
                     lock .lock();
                     if (instance == null) {
                            instance = new Singleton();
                     }
                     lock .unlock();
              }
              return instance ;
       }
}
```

第 4 章
线程安全的集合类

知识是学出来的，能力是练出来的。



一起看一下 Java 中常用的线程安全的集合类有哪些，并且来一起认识它。

4.1 java.util.Hashtable

Hashtable 和 HashMap 一样，Hashtable 也是一个散列表，它存储的内容是键值对（key-value）映射。

Hashtable 继承于 Dictionary，它实现了 Map、Cloneable、java.io.Serializable 接口。

Hashtable 的实例有两个参数影响其性能：初始容量和加载因子。容量是哈希表中桶的数量，初始容量就是哈希表创建时的容量。注意，哈希表的状态为 open。在发生“哈希冲突”的情况下，单个桶会存储多个条目，这些条目必须按顺序搜索。加载因子是对哈希表在其容量自动增加之前可以达到多满的一个尺度。初始容量和加载因子这两个参数只是对该实现的提示。关于何时以及是否调用 rehash 方法的具体细节则依赖于该实现。

通常，默认加载因子是 0.75，这是在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查找某个条目的时间（在大多数 Hashtable 操作中，包括 get 和 put 操作，都反映了这一点）。

Hashtable 的函数都是同步的，这意味着它是线程安全的。它的 key、value 都不可以为 null。此外，Hashtable 中的映射不是有序的。

那么 Hashtable 如何保证线程安全性的呢？下面是 Hashtable 的源码：

```
public class Hashtable <K, V> extends Dictionary<K, V>
 implements Map<K, V>, Cloneable, Serializable
{
    private transient Entry<?, ?>[] table ;
    public synchronized V get(Object key) {
            Entry tab[] = table ;
           ..... //此处省略，具体的实现请参考 jdk 实现
    }
    public synchronized V put( K key, V value) {
            ..... //具体实现省略，请参考 jdk 实现
    }
    public synchronized V remove(Object key) {
            ..... //具体实现省略，请参考 jdk 实现
    }
    public synchronized void putAll(Map<? extends K, ? extends V > t) {
       for (Map.Entry<? extends K , ? extends V> e : t.entrySet())
           put(e.getKey(), e.getValue());
    }
    public synchronized void clear() {
     ..... //具体实现省略，请参考 jdk 实现
    }
    //...
}
```

上面是 Hashtable 提供的几个主要方法，包括 get()、put()、remove() 等。注意到每个方法本身都是 synchronized 的，不会出现两个线程同时对数据进行操作的情况，因此保证了线程安全性。

Hashtable 的主要对外接口：

- clear() 的作用是清空 Hashtable。它是将 Hashtable 的 table 数组的值全部设为 null。
- contains() 和 containsValue() 的作用都是判断 Hashtable 是否包含“值(value)”。
- containsKey() 的作用是判断 Hashtable 是否包含 key。
- elements() 的作用是返回“所有 value”的枚举对象。
- Enumerator 的作用是提供了“通过 elements()遍历 Hashtable 的接口”。因为，它同时实现了“Enumerator 接口”和“Iterator 接口”。
- entrySet()、keySet()、keys()、values()的实现方法和 elements()差不多，而且源码中已经明确地给出了注释。这里就不再做过多说明了。
- get() 的作用就是获取 key 对应的 value，没有的话返回 null。
- put() 的作用是对外提供接口，让 Hashtable 对象可以通过 put()将“key-value”添加到

Hashtable 中。

- putAll() 的作用是将“Map(t)”中的全部元素逐一添加到 Hashtable 中。
- remove() 的作用就是删除 Hashtable 中键为 key 的元素。

Hashtable 实现的 Cloneable 接口，即实现了 clone()方法。clone()方法的作用很简单，就是克隆一个 Hashtable 对象并返回。

Hashtable 实现的 Serializable 接口分别实现了串行读取、写入功能。

Hashtable 的使用实例如下：

```
public static void main(String[] args) {
      Hashtable<String, Integer> numbers = new Hashtable<String, Integer>();
      numbers.put( "one", 1);
      numbers.put( "two", 2);
      numbers.put( "three", 3);
      Integer n = numbers.get( "two");
       if (n != null) {
            System. out.println("two = " + n);
      }
}
```

4.2 java.util.concurrent.ConcurrentHashMap

ConcurrentHashMap 继承于 AbstractMap，实现了 Map、java.io.Serializable 接口。

Concurrent.ConcurrentHashMap 是并发编程大师 Doug Lea 的作品。这是 HashMap 的线程安全版，同 Hashtable 相比，ConcurrentHashMap 不仅保证了访问的线程安全性，而且在效率上与 Hashtable 相比，有较大的提高。ConcurrentHashMap 允许多个修改操作并发进行，其关键在于使用了锁分离技术，即代码块锁，而不是方法锁。它使用了多个锁来控制对 hash 表的不同部分进行的修改。ConcurrentHashMap 内部使用段（Segment）来表示这些不同的部分，每个段其实就是一个小的 hash table，它们有自己的锁（由 ReentrantLock 来实现的）。只要多个修改操作发生在不同的段上，它们就可以并发进行。

ConcurrentHashMap 的部分源码如下：

```
public class ConcurrentHashMap<K,V> extends AbstractMap<K,V>
   implements ConcurrentMap<K,V>, Serializable {
.....
   static class Segment<K,V> extends ReentrantLock implements Serializable {
      private static final long serialVersionUID = 2249069246763182397L;
```

```
        final float loadFactor ;
        Segment( float lf) { this.loadFactor = lf; }
    }
}
```

ConcurrentHashMap 常用的主要对外接口：

- clear() 的作用是清空 ConcurrentHashMap。
- contains() 和 containsValue() 的作用都是判断 ConcurrentHashMap 是否包含“值(value)”。
- containsKey() 的作用是判断 ConcurrentHashMap 是否包含 key。
- elements() 的作用是返回“所有 value”的枚举对象。
- Enumerator 的作用是提供了“通过 elements()遍历 Hashtable 的接口”和“通过 entrySet() 遍历 ConcurrentHashMap 的接口”。因为，它同时实现了“Enumerator 接口”和“Iterator 接口”。
- entrySet()、keySet()、keys()、values()的实现方法和 elements()差不多，而且源码中已经明确地给出了注释，这里就不再做过多说明了。
- get() 的作用就是获取 key 对应的 value，没有的话返回 null。
- put() 的作用是对外提供接口，让 ConcurrentHashMap 对象可以通过 put()将“key-value”添加到 ConcurrentHashMap 中。
- putAll() 的作用是将“Map(t)”中的全部元素逐一添加到 ConcurrentHashMap 中。
- remove() 的作用就是删除 ConcurrentHashMap 中键为 key 的元素。

ConcurrentHashMap 实现的 Serializable 接口分别实现了串行读取、写入功能。

ConcurrentHashMap 使用实例如下：

```
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<String,
Integer>();
        map.put( "one", 1);
        map.put( "two", 2);
        map.put( "three", 3);
        System. out.println(map.get("two" ));
         if (map.containsKey("one" ) && map.get("one").equals(1)) {
             map.remove( "one");
        }
    }
```

4.3 java.util.concurrent.CopyOnWriteArrayList

CopyOnWriteArrayList 中的 set、add、remove 等方法，都使用了 ReentrantLock 的 lock()来加锁，unlock()来解锁。当增加元素的时候使用 Arrays.copyOf()来拷贝副本，在副本上增加元素，然后改变原引用指向副本。读操作不需要加锁，而写操作类实现中对其进行了加锁。因此，CopyOnWriteArrayList 类是一个线程安全的 List 接口的实现，这对于读操作远远多于写操作的应用非常适合。特别是在并发情况下，可以提供高性能的并发读取，并且保证读取的内容一定是正确的，不受多线程并发问题影响的。

且看 CopyOnWriteArrayList 部分源码，其中的一个简单 add 方法的实现：

```
public class CopyOnWriteArrayList<E>
 implements List<E>, RandomAccess, Cloneable, Serializable
{
private volatile transient Object[] array;
final transient ReentrantLock lock = new ReentrantLock();
public boolean add (E paramE)
{
   ReentrantLock localReentrantLock = this.lock ;//加锁的安全机制
   localReentrantLock.lock();
   try
   {
     Object[] arrayOfObject1 = getArray();
     int i = arrayOfObject1. length;
     Object[] arrayOfObject2 = Arrays. copyOf(arrayOfObject1, i + 1);
     arrayOfObject2[i] = paramE;
     setArray(arrayOfObject2);
     int j = 1;
     return j;
    }
    finally
    {//finally 里面释放锁
     localReentrantLock.unlock();
    }
  }
  //get 方法没有加锁
  private E get(Object[] a, int index) {
      return (E) a[index];
```

```
    }
    //……
}
```

CopyOnWriteArrayList 主要的常用的方法有：

- boolean add(E e)，在 list 末尾添加一个元素。
- void add(int index，E element)，在指定位置添加元素。
- boolean addAll(Collection<? extends E> c) ，在尾部增加整个 Collection 对象。
- boolean addAll(int index，Collection<? extends E> c)，在索引指定的位置插入 Collection 对象并移动原位置（如果有的话）的对象。
- void clear()，清空 list 集合。
- Object clone()，由于实现了 Cloneable 接口，所以支持克隆。
- boolean contains(Object o)，是否包含指定元素。
- E get(int index)，获得指定位置的元素。
- boolean isEmpty()，半段集合是否是空的。
- Iterator<E>，iterator()的作用是提供了“通过 elements()遍历。因为，它同时实现了“Enumerator 接口”和“Iterator 接口”。
- ListIterator<E>，listIterator()的作用是提供了“通过 elements()遍历。因为，它同时实现了“Enumerator 接口”和“Iterator 接口”。
- E remove，删除指定元素。
- <T> T[] toArray(T[] a)，将集合转换成数组。

CopyOnWriteArrayList 简单的使用实例如下：

```
public static void main(String[] args) {
     CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<String>();
     list.add( "one");
     list.add( "three");
     list.add(1, "two");//list 下标是从零开始的
     System. out.println(list.get(1));
      if (list.contains("three" )) {
           Iterator<String> value = list.iterator();
            while(value.hasNext()) {
                 System. out.println(value.next());
           }
     }
}
```

运行结果如下：

```
two
one
two
three
```

4.4 java.util.concurrent.CopyOnWriteArraySet

CopyOnWriteArraySet 是在 CopyOnWriteArrayList 的基础上使用了 Java 的装饰模式，很多方法如：储存介质使用了 CopyOnWriteArrayList 来存储数据，remove 方法调用 CopyOnWriteArrayList 的 remove 方法，add 方法调用了 CopyOnWriteArrayList 的 addIfAbsent 方法。所以 CopyOnWriteArrayList 的实现原理适用 CopyOnWriteArraySet。

且看 CopyOnWriteArraySet 的一段源码：

```
public class CopyOnWriteArraySet<E> extends AbstractSet<E>
 implements Serializable
{
 //看的出来大部分都是调用 CopyOnWriteArrayList 的 API 实现的
 private final CopyOnWriteArrayList<E> al ;

 public CopyOnWriteArraySet()
 {
   this. al = new CopyOnWriteArrayList();
 }
 public void clear()
 {
   this. al.clear();
 }

 public boolean remove(Object paramObject)
 {
   return this.al .remove(paramObject);
 }

 public boolean add(E paramE)
 {
   return this.al .addIfAbsent(paramE);
 }
```

```
    ……//大部分代码皆是如此
}
```

CopyOnWriteArraySet 主要的常用方法有：

- boolean add(E e)，在 list 末尾添加一个元素。
- boolean addAll(Collection<? extends E> c)，在尾部增加整个 Collection 对象。
- void clear()，清空 list 集合。
- boolean contains(Object o)，是否包含指定元素。
- boolean isEmpty()，判断集合是否空的。
- Iterator<E>，iterator()的作用是提供了对 elements()遍历。因为，它同时实现了 “Enumerator 接口” 和 “Iterator 接口” 。
- E remove，删除指定元素。
- <T> T[] toArray(T[] a)，将集合转换成数组。

CopyOnWriteArraySet 简单的使用实例如下：

```
CopyOnWriteArraySet<String> list = new CopyOnWriteArraySet<String>();
list.add( "one");
list.add( "three");
if (list.contains("three" )) {
        Iterator<String> value = list.iterator();
        while(value.hasNext()) {
             System. out.println(value.next());
        }
}
```

Java 里面 List 和 Set 的相同和区别同样适用于 CopyOnWriteArrayList 和 CopyOnWriteArraySet 不同之处是后两者是线程安全机制。

4.5 CopyOnWrite 机制介绍

CopyOnWrite 容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行 Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对 CopyOnWrite 容器进行并发地读，而不需要加锁，因为当前容器不会添加任何元素。所以 CopyOnWrite 容器也是一种读写分离的思想，读和写不同的容器。

我们从 CopyOnWriteArrayList 之前的分析和源码中可以看得出来，ArrayList 里添加元素，可

以发现在添加的时候是需要加锁的，否则多线程写的时候会 Copy 出 N 个副本出来。

读的时候不需要加锁，如果读的时候有多个线程正在向 ArrayList 添加数据，还是会读到旧的数据，因为写的时候不会锁住旧的 ArrayList。

```
public E get(int index) {
    return get(getArray(), index);
}
```

JDK 中并没有提供 CopyOnWriteMap，我们可以参考 CopyOnWriteArrayList 来实现一个，基本代码如下：

```
import java.util.Collection;
import java.util.Map;
import java.util.Set;

public class CopyOnWriteMap<K, V> implements Map<K, V>, Cloneable {
    private volatile Map<K, V> internalMap;

    public CopyOnWriteMap() {
        internalMap = new HashMap<K, V>();
    }

    public V put(K key, V value) {

        synchronized (this) {
            Map<K, V> newMap = new HashMap<K, V>(internalMap);
            V val = newMap.put(key, value);
            internalMap = newMap;
            return val;
        }
    }

    public V get(Object key) {
        return internalMap.get(key);
    }

    public void putAll(Map<? extends K, ? extends V> newData) {
        synchronized (this) {
            Map<K, V> newMap = new HashMap<K, V>(internalMap);
            newMap.putAll(newData);
```

```
                internalMap = newMap;
            }
        }
    }
```

实现很简单，只要了解了 CopyOnWrite 机制，我们可以实现各种 CopyOnWrite 容器，并且在不同的应用场景中使用。

CopyOnWrite 的应用场景：CopyOnWrite 并发容器用于读多写少的并发场景，比如白名单，黑名单，商品类目的访问和更新等等场景。假如我们有一个搜索网站，用户在这个网站的搜索框中，输入关键字搜索内容，但是某些关键字不允许被搜索。这些不能被搜索的关键字会被放在一个黑名单当中，黑名单每天晚上更新一次。当用户搜索时，会检查当前关键字在不在黑名单当中，如果在，则提示不能搜索。

代码很简单，但是使用 CopyOnWriteMap 需要注意两件事情：

（1）减少扩容开销。根据实际需要，初始化 CopyOnWriteMap 的大小，避免写时 CopyOnWriteMap 扩容的开销。

（2）使用批量添加。因为每次添加，容器每次都会进行复制，所以减少添加次数，可以减少容器的复制次数。如使用上面代码里的 addBlackList 方法。

CopyOnWrite 的缺点：

（1）内存占用问题。因为 CopyOnWrite 的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意：在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说 200MB 左右，那么再写入 100MB 数据进去，内存就会占用 300MB，那么这个时候很有可能造成频繁的 Yong GC 和 Full GC。之前我们系统中使用了一个服务由于每晚使用 CopyOnWrite 机制更新大对象，造成了每晚 15s 的 Full GC，应用响应时间也随之变长。针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗。

（2）数据一致性问题。CopyOnWrite 容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的数据马上能读到，请不要使用 CopyOnWrite 容器。

4.6 Vector

Vector 是矢量队列，通过数组保存数据。它继承于 AbstractList，实现了 List、RandomAccess、Cloneable 这些接口。

Vector 继承了 AbstractList，实现了 List；所以，它是一个队列，支持相关的添加、删除、修

改、遍历等功能。

Vector 实现了 RandmoAccess 接口，即提供了随机访问功能。RandmoAccess 是 Java 中用来被 List 实现，为 List 提供快速访问功能的。在 Vector 中，我们可以通过元素的序号快速获取元素对象，这就是快速随机访问。

Vector 实现了 Cloneable 接口，即实现 clone()函数。它能被克隆。

和 ArrayList 不同，Vector 中的操作是线程安全的，它是利用 synchronized 同步显示锁的方法锁的机制实现，实现安全机制类似 Hashtable 。

下面看一下的 Vector 的源码，注意方法锁：

```
public class Vector<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io. Serializable
{

    protected Object[] elementData;
    public synchronized void addElement(E obj) {
        modCount++;
        ensureCapacityHelper( elementCount + 1);
        elementData[elementCount ++] = obj;
    }
       public synchronized boolean removeElement(Object obj) {
          modCount++;
          int i = indexOf(obj);
          if (i >= 0) {
              removeElementAt(i);
              return true ;
          }
          return false ;
       }
    //同步方法锁
    public synchronized E get( int index) {
        if (index >= elementCount )
            throw new ArrayIndexOutOfBoundsException(index);

        return elementData(index);
    }
.....
}
```

上面是 Vector 提供的几个主要方法，包括 addElement()、removeElement()、get() 等。注意到每个方法本身都是 synchronized 的，不会出现两个线程同时对数据进行操作的情况，因此保证了线程安全性。

Vector 常用的方法如下，与其他集合类的方法大同小异，我们就看几个例子吧：

- void addElement(E obj)，添加元素。
- void clear()，清空集合。
- Object clone()，集合克隆。
- boolean contains(Object o)，是否包含某一个元素。
- E firstElement()，获得第一个元素。
- int indexOf，得到一个元素的下标。
- boolean isEmpty()，判断集合是否空的。
- Iterator<E> iterator()，获得 iterator 集合对象。
- E lastElement()，获得最后一个元素。
- boolean removeElement，删除一个元素。
- Object[] toArray，得到一个数组结果集。

Vector 的使用实例：

```
public static void main(String[] args) {
    Vector<String> list = new Vector<String>();
    list.addElement( "one");
    list.addElement( "two");
    list.addElement( "three");
    list.removeElement ("two" );
     if (list.contains("three" )) {
         Iterator<String> value = list.iterator();
          while(value.hasNext()) {
              System. out.println(value.next());
          }
    }
}
```

4.7 常用的 StringBuffer 与 StringBuilder

在编写 JAVA 代码的过程中有时要频繁地对字符串进行拼接，如果直接用“+”拼接的话会建立很多的 String 型对象，严重的话会对服务器资源和性能造成不小的影响；而使用 StringBuilder

和 StringBuffer 能解决以上问题。而 StringBuffer 是线程安全的，而 StringBuilder 不是线程安全的。

来看一下 StringBuffer 的一些源码：

```
public final class StringBuffer
    extends AbstractStringBuilder
    implements java.io.Serializable, CharSequence
{
    //方法同步锁
    public synchronized StringBuffer append(String str) {
        toStringCache = null ;
        super.append(str);
        return this ;
    }
    //方法同步锁
    public synchronized String toString() {
        if (toStringCache == null) {
            toStringCache = Arrays.copyOfRange( value, 0, count);
        }
        return new String(toStringCache, true);
    }
.....
}
```

而 StringBuilder 的源码一部分如下：

```
public final class StringBuilder
    extends AbstractStringBuilder
    implements java.io.Serializable, CharSequence
{
    public StringBuilder append(String str) {
        super.append(str);
        return this ;
    }
    public String toString() {
        return new String(value, 0, count);
    }
.....
}
```

而与之对应的 StringBuilder 是线程不安全的，也就是没有加 **synchronized** 锁。所以不难看出我们在高并发的情况下，如果不需要考虑数据安全问题的情况下，尽量使用 StringBuffer，由于没

有资源等待的情况，肯定执行效率和性能会高很多。所以是使用 StringBuffer 还是使用 StringBuilder 要视情况而定。

下面我们对线程安全做个小结：

通过第 3 章、第 4 章的学习我们不难看出，在掌握了线程的安全原理和锁的机制的情况下再去理解和查看 Java 里面所提供的线程的安全类，我们就知道怎么回事了，所以我们在日常的 Web 开发过程中也可以适当地根据原理和业务写出自己的线程安全类。最后关于执行效率总结一句：有关代码执行效率，没有加锁的、不需要同步安全的代码执行效率最高>方法块锁>类锁和方法锁，同时要注意读写分离的业务场景。知识是学出来的，能力是练出来的，学习了要熟练用之于实践。

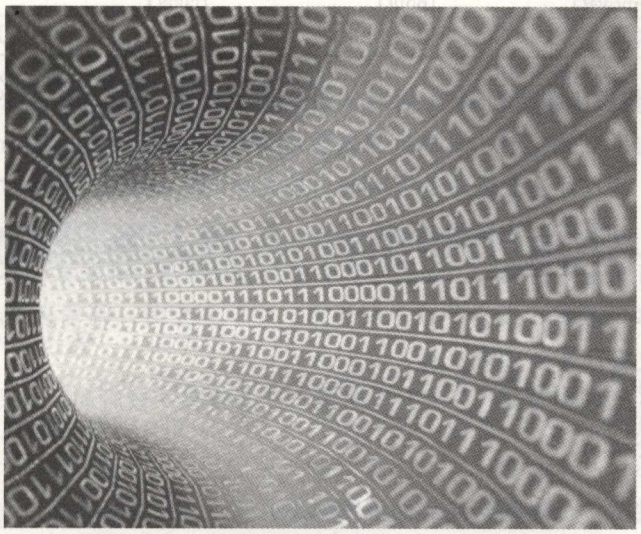
第2部分

线程并发晋级之高级部分

第 5 章

多线程之间交互：线程阀

投入多少，收获多少；参与多深，领悟多深。



线程阀是一种线程与线程之间相互制约和交互的机制。

5.1 阻塞队列 BlockingQueue

先理解 Queue、Deque、BlockingQueue 的概念。

- Queue（队列）：用于保存一组元素，不过在存取元素的时候必须遵循先进先出原则。队列是一种特殊的线性表，它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。在队列这种数据结构中，最先插入的元素将是最先被删除的元素；反之最后插入的元素将是最后被删除的元素，因此队列又称为“先进先出”（FIFO—first in first out）的线性表。
- Deque（双端队列）： 两端都可以进出的队列。当我们约束从队列的一端进出队时，就

形成了另外一种存取模式，它遵循先进后出原则，这就是栈结构。双端队列主要是用于栈操作。使用栈结构让操作有可追溯性（如 Windows 窗口地址栏内的路径前进栈、后退栈）。

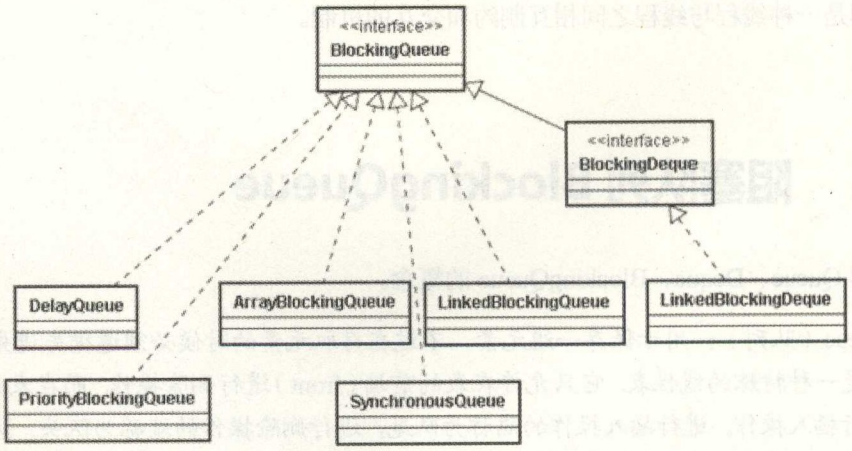
阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空；当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿取元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿取元素。

阻塞队列提供了 4 种处理方法:

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除方法	remove()	poll()	take()	poll(time,unit)
检查方法	element()	peek()	不可用	不可用

- 抛出异常: 是指当阻塞队列满时，再往队列里插入元素，会抛出 IllegalStateException("Queue full")异常。当队列为空时，从队列里获取元素时会抛出 NoSuchElementException 异常 。
- 返回特殊值: 插入方法会返回是否成功，若成功则返回 true。移除方法，则是从队列里拿出一个元素，如果没有则返回 null。
- 一直阻塞: 当阻塞队列满时，如果生产者线程往队列里 put 元素，队列会一直阻塞生产者线程，直到拿到数据，或者响应中断退出。当队列空时，消费者线程试图从队列里 take 元素，队列也会阻塞消费者线程，直到队列可用。
- 超时退出: 当阻塞队列满时，队列会阻塞生产者线程一段时间，如果超过一定的时间，生产者线程就会退出。

最新 JDK 中提供了 7 个阻塞队列，如下图所示:



BlockingQueue 常用的方法有如下 5 种，更多方法请查询 API。

（1）add(anObject)：把 anObject 加到 BlockingQueue 里，即如果 BlockingQueue 可以容纳，则返回 true，否则抛出异常。

（2）offer(anObject)：表示如果可能的话，将 anObject 加到 BlockingQueue 里，即如果 BlockingQueue 可以容纳，则返回 true，否则返回 false。

（3）put(anObject)：把 anObject 加到 BlockingQueue 里，如果 BlockQueue 没有空间，则调用此方法的线程被阻断，直到 BlockingQueue 里面有空间时再继续。

（4）poll(time)：取走 BlockingQueue 里排在首位的对象，若不能立即取出，则可以等 time 参数规定的时间，取不到时返回 null。

（5）take()：取走 BlockingQueue 里排在首位的对象，若 BlockingQueue 为空，阻断进入等待状态直到 Blocking 有新的对象被加入为止。

其中：BlockingQueue 不接受 null 元素。试图 add、put 或 offer 一个 null 元素时，某些实现会抛出 NullPointerException 异常。null 被用作指示 poll 操作失败的警戒值。

5.2 数组阻塞队列 ArrayBlockingQueue

ArrayBlockingQueue 是一个由数组支持的有界的阻塞队列。此队列按 FIFO（先进先出）原则对元素进行排序。队列的*头部*是在队列中存在时间最长的元素。队列的*尾部*是在队列中存在时间最短的元素。新元素插入到队列的尾部，队列获取操作则是从队列头部开始获得元素。

这是一个典型的“有界缓存区”，固定大小的数组在其中保持生产者插入的元素和使用者提取的元素。一旦创建了这样的缓存区，就不能再增加其容量。试图向已满队列中放入元素会导致操作受阻塞；试图从空队列中提取元素将导致类似阻塞。

此类支持对等待的生产者线程和使用者线程进行排序的可选公平策略。默认情况下，不保证是这种排序。然而，通过将公平性（fairness）设置为 true 而构造的队列允许按照 FIFO 顺序访问线程。公平性通常会降低吞吐量，但也减少了可变性和避免了“不平衡性”。

我们先看一下 ArrayBlockingQueue 的部分源码，理解一下 ArrayBlockingQueue 的实现原理和机制。

```
public class ArrayBlockingQueue <E> extends AbstractQueue<E>
        implements BlockingQueue<E>, java.io.Serializable {

    //数组的储存结构
    final Object[] items;
   //锁采用的机制
    final ReentrantLock lock;
    public ArrayBlockingQueue( int capacity, boolean fair) {
```

```
        if (capacity <= 0)
            throw new IllegalArgumentException();
        this.items = new Object[capacity];
        //通过将公平性 (fairness) 设置为 true 而构造的队列允许按照 FIFO 顺序访问线程
        lock = new ReentrantLock(fair);
        notEmpty = lock .newCondition();
        notFull =  lock .newCondition();
    }
    public boolean offer(E e) {
        checkNotNull(e);
        //使用 ReentrantLock 锁机制
        final ReentrantLock lock = this.lock;
        lock.lock();//加锁
        try {
            if (count == items.length)
                return false ;
            else {
                enqueue(e);
                return true ;
            }
        } finally {
            lock.unlock();//释放锁
        }
    }
    private void enqueue(E x) {
        final Object[] items = this.items;
        items[ putIndex] = x;//通过数组进行储存
        if (++putIndex == items.length)
            putIndex = 0;
        count++;
        notEmpty.signal();
    }
.....
}
```

使用实例是：

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
/*
```

```
* 现有的程序代码模拟产生了16个日志对象，并且需要运行16s 才能打印完这些日志，
* 请在程序中增加4个线程去调用 parseLog()方法来分头打印这16个日志对象，
* 程序只需要运行4s 即可打印完这些日志对象。
*/
public class BlockingQueueTest {
    public static void main(String[] args) throws Exception {
        // 新建一个等待队列
        final BlockingQueue<String> bq = new ArrayBlockingQueue<String>(16);
        // 4个线程
        for (int i = 0; i < 4; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    while (true) {
                        try {
                            String log = (String) bq.take();
                            parseLog(log);
                        } catch (Exception e) {
                        }
                    }
                }
            }).start();
        }
        for (int i = 0; i < 16; i++) {
            String log = (i + 1) + " --> ";
            bq.put(log); // 将数据存到队列里!
        }
    }
    // parseLog 方法内部的代码不能改动
    public static void parseLog(String log) {
        System.out.println(log + System.currentTimeMillis());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

5.3 链表阻塞队列 LinkedBlockingQueue

LinkedBlockingQueue 是基于链表的阻塞队列，同 ArrayListBlockingQueue 类似，其内部也维持着一个数据缓冲队列（该队列由一个链表构成），当生产者往队列中放入一个数据时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回。只有当队列缓冲区达到最大值缓存容量时（LinkedBlockingQueue 可以通过构造函数指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒。反之，对于消费者这端的处理也基于同样的原理。而 LinkedBlockingQueue 之所以能够高效地处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。

作为开发者，我们需要注意的是，如果构造一个 LinkedBlockingQueue 对象，而没有指定其容量大小，LinkedBlockingQueue 会默认一个类似无限大小的容量（Integer.MAX_VALUE），这样的话，如果生产者的速度一旦大于消费者的速度，也许还没有等到队列满阻塞产生，系统内存就有可能已被消耗殆尽了。

我们先看一下 LinkedBlockingDeque 的部分源码，理解一下 ArrayBlockingQueue 的实现原理和机制。

```
public class LinkedBlockingDeque <E>
    extends AbstractQueue<E>
    implements BlockingDeque<E>, java.io.Serializable {
    final ReentrantLock lock = new ReentrantLock();//线程安全
    /**
     * @throws NullPointerException {@inheritDoc}
     */
    public boolean offerLast(E e) {
        if (e == null) throw new NullPointerException();
        Node<E> node = new Node<E>(e);//每次插入后都将动态地创建链接节点
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            return linkLast(node);
        } finally {
            lock.unlock();
        }
    }
    public boolean offer(E e) {
        return offerLast(e);
```

```
    }
    public boolean add(E e) {
        addLast(e);
        return true;
    }
    public void addLast(E e) {
        if (!offerLast(e))
            throw new IllegalStateException("Deque full");
    }
    public E removeFirst() {
        E x = pollFirst();
        if (x == null) throw new NoSuchElementException();
        return x;
    }
    public E pollFirst() {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            return unlinkFirst();
        } finally {
            lock.unlock();
        }
    }
.....
}
```

这里使用实例与上一节中 ArrayBlockingQueue 的例子一样。不同的是，将 ArrayBlockingQueue 的例子换成 LinkedBlockingQueue 即可：

```
final BlockingQueue<String> bq = new ArrayBlockingQueue<String>(16);
```

换成：

```
final BlockingQueue<String> bq = new LinkedBlockingQueue<String>(16);
```

5.4 优先级阻塞队列 PriorityBlockingQueue

PriorityBlockingQueue 是一个支持优先级排序的无界阻塞队列（优先级的判断通过构造函数传入的 Compator 对象来决定），但需要注意的是 PriorityBlockingQueue 并不会阻塞数据生产者，而只会在没有可消费的数据时，阻塞数据的消费者。因此使用的时候要特别注意，生产者生产数据的速度绝对不能快于消费者消费数据的速度，否则时间一长，会最终耗尽所有的可用堆内存空间。在实现 PriorityBlockingQueue 时，内部控制线程同步的锁采用的是公平锁。

先看一下 PriorityBlockingQueue 的部分源码，理解一下 PriorityBlockingQueue 的实现原理和机制：

```
public class PriorityBlockingQueue <E> extends AbstractQueue<E>
    implements BlockingQueue<E>, java.io.Serializable {
    private final ReentrantLock lock ;//说明本类使用一个 lock 来同步读写等操作
    private transient Comparator<? super E> comparator;
     // 使用指定的初始容量创建一个 PriorityBlockingQueue，并根据指定的比较器对其元素进行排序。
    public PriorityBlockingQueue( int initialCapacity,
                                  Comparator<? super E> comparator) {
        if (initialCapacity < 1)
            throw new IllegalArgumentException();
        this.lock = new ReentrantLock();
        this.notEmpty = lock.newCondition();
        this.comparator = comparator;
        this.queue = new Object[initialCapacity];
    }
     public E poll() {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            return dequeue();
        } finally {
            lock.unlock();
        }
    }
.....
}
```

5.5 延时队列 DelayQueue

DelayQueue 是一个支持延时获取元素的使用优先级队列的实现的无界阻塞队列。队列中的元素必须实现 Delayed 接口和 Comparable 接口，也就是说 DelayQueue 里面的元素必须有 public int compareTo(To)和 long getDelay(TimeUnit unit)方法存在，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。我们可以将 DelayQueue 运用在以下场景中：

- 缓存系统的设计。可以用 DelayQueue 保存缓存元素的有效期，使用一个线程循环查询 DelayQueue，一旦能从 DelayQueue 中获取元素时，表示缓存有效期到了。
- 定时任务调度。使用 DelayQueue 保存当天将会执行的任务和执行时间，一旦从 DelayQueue 中获取到任务就开始执行，比如 TimerQueue 就是使用 DelayQueue 实现的。

我们来看一下 DelayQueue 的源码，来理解一下 DelayQueue 的实现原理和机制：

```
//可以看出来 E 元素必须继承 Delayed 和而 Delayed 又继承 Comparable。
public class DelayQueue<E extends Delayed> extends AbstractQueue<E>
    implements BlockingQueue<E> {

    private final transient ReentrantLock lock = new ReentrantLock();//安全锁机制
    private final PriorityQueue<E> q = new PriorityQueue<E>();//PriorityQueue 来存
取元素
    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            for (;;) {
                E first = q.peek();
                if (first == null)
                    available.await();
                else {
                   //根据元素的 Delay 进行判断
                    long delay = first.getDelay(NANOSECONDS);
                    if (delay <= 0)
                        return q.poll();
                    first = null; // don't retain ref while waiting
                    if (leader != null)
                      //没到时间阻塞等待
```

```
                        available.await();
                    else {
                        Thread thisThread = Thread. currentThread();
                        leader = thisThread;
                        try {
                            available.awaitNanos(delay);
                        } finally {
                            if (leader == thisThread)
                                leader = null ;
                        }
                    }
                }
            }
        } finally {
            if (leader == null && q.peek() != null)
                available.signal();
            lock.unlock();
        }
    }
    ......
}
```

我们来看一下 DelayQueue 的使用实例：

（1）实现一个 Student 对象作为 DelayQueue 的元素必须实现 Delayed 接口的两个方法。

```
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;
public class Student implements Delayed { //必须实现 Delayed 接口
    private String name ;
    private long submitTime ;// 交卷时间
    private long workTime ;// 考试时间
    public String getName() {
        return this .name + " 交卷,用时" + workTime;
    }
    public Student(String name, long submitTime) {
        this.name = name;
        this.workTime = submitTime;
        this.submitTime = TimeUnit.NANOSECONDS.convert(submitTime, TimeUnit.
MILLISECONDS ) + System.nanoTime ();
```

```
            System. out.println(this.name + " 交卷,用时" + workTime);
        }
        //必须实现 getDelay 方法
        public long getDelay(TimeUnit unit) {
//          返回一个延迟时间
            return unit.convert(submitTime - System.nanoTime (), unit.NANOSECONDS );
        }
        //必须实现 compareTo 方法
        public int compareTo(Delayed o) {
//          比较的方法
            Student that = (Student) o;
            return submitTime > that.submitTime ? 1 : ( submitTime < that.
submitTime ? -1 : 0);
        }
    }
```

（2）执行运行类如下：

```
    package demo.thread;
    import java.util.concurrent.DelayQueue;
    public class DelayQueueTest {
        public static void main(String[] args) throws Exception {
            // 新建一个等待队列
            final DelayQueue<Student> bq = new DelayQueue<Student>();
            for (int i = 0; i < 5; i++) {
                Student student = new Student("学生" +i,Math.round((Math.
random()*10+i)));
                bq.put(student); // 将数据存到队列里!
            }
            //获取但不移除此队列的头部；如果此队列为空，则返回 null。
            System. out.println("bq.peek()"+bq.peek().getName());
            //获取并移除此队列的头部，在可从此队列获得到期延迟的元素，或者到达指定的等待时
间之前一直等待（如有必要）。
            //poll(long timeout, TimeUnit unit) 大家可以试一试这个方法
        }
    }
```

运行结果如下：每次运行结果都不一样，我们获得永远是队列里面的第一个元素。

```
学生0 交卷,用时8
学生1 交卷,用时6
```

```
学生2 交卷,用时10
学生3 交卷,用时10
学生4 交卷,用时9
bq.peek()学生1 交卷,用时6
```

读者可以慢慢地在以后的工作当中体会 DelayQueue 的用法。

5.6 同步队列 SynchronousQueue

SynchronousQueue 是一个不存储元素的阻塞队列。每一个 put 操作必须等待一个 take 操作，否则不能继续添加元素。SynchronousQueue 可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身并不存储任何元素，非常适合于传递性场景，比如在一个线程中使用的数据，传递给另外一个线程使用，SynchronousQueue 的吞吐量高于 LinkedBlockingQueue 和 ArrayBlockingQueue。

声明一个 SynchronousQueue 有两种不同的方式，它们之间有着不太一样的行为。公平模式和非公平模式的区别：如果采用公平模式，SynchronousQueue 会采用公平锁，并配合一个 FIFO 队列来阻塞多余的生产者和消费者，从而体系整体的公平策略；但如果是非公平模式（SynchronousQueue 默认），SynchronousQueue 采用非公平锁，同时配合一个 LIFO 队列来管理多余的生产者和消费者，而后一种模式，如果生产者和消费者的处理速度有差距，则很容易出现饥渴的情况，即可能有某些生产者或者是消费者的数据永远都得不到处理。

来看部分 SynchronousQueue 的源码，理解一下 SynchronousQueue 的实现原理和机制：

```
public class SynchronousQueue <E> extends AbstractQueue<E>
    implements BlockingQueue<E>, java.io.Serializable {
    public SynchronousQueue( boolean fair) {//安全的两种机制
        transferer = fair ? new TransferQueue<E>() : new TransferStack<E>();
    }
    public E take() throws InterruptedException {
        E e = transferer.transfer(null, false, 0);
        if (e != null)
            return e;
        Thread. interrupted();
        throw new InterruptedException();
    }
.....
}
```

因为 SynchronousQueue 没有内部容量，所以只提供以下方法：

- isEmpty(): 始终返回 true。
- size(): 始终返回 0。
- remainingCapacity(): 始终返回 0。
- clear(): 不执行任何操作。
- remove(Object o): 始终返回 false。
- containsAll(Collection<?> c): 除非给定 collection 为空，否则返回 false。
- removeAll(Collection<?> c): 始终返回 false。
- retainAll(Collection<?> c): 始终返回 false。
- peek(): 始终返回 null。
- iterator(): 返回一个空迭代器，其中 hasNext 始终返回 false。
- toArray(): 返回一个 0 长度的数组。

SynchronousQueue 简单的使用实例如下：

```
import java.util.concurrent.Semaphore;
import java.util.concurrent.SynchronousQueue;
/*
 * 现成程序中的 Test1类中的代码在不断地产生数据，
 * 然后交给 TestDo.doSome()方法去处理，
 * 就好像生产者在不断地产生数据，消费者在不断消费数据。
 *
 * 请将程序改造成有10个线程来消费生成者产生的数据，这些消费者都调用 TestDo.doSome()方法去进行处理，故每个消费者都需要一秒才能处理完，程序应保证这些消费者线程依次有序地消费数据，只有上一个消费者消费完后，下一个消费者才能消费数据，下一个消费者是谁都可以，但要保证这些消费者线程拿到的数据是有顺序的。
 */
public class SynchronousQueueTest {
        public static void main(String[] args) {
                System. out.println("begin:" +     (System.currentTimeMillis ()     /
1000));
                // 定义一个 Synchronous
                final SynchronousQueue<String> sq = new SynchronousQueue<String>();
                // 定义一个数量为1的信号量，其作用相当于一个互斥锁
                final Semaphore sem = new Semaphore(1);
                for (int i = 0; i < 10; i++) {
                        new Thread(new Runnable() {
                                public void run() {
                                        try {
```

```
                                    sem.acquire();
                                    String input = sq.take();
                                    String output = TestDo.doSome(input);
                                    System. out.println(Thread.currentThread()
.getName() + ":" + output);
                                    sem.release();
                                } catch (InterruptedException e) {
                                    e.printStackTrace();
                                }
                            }
                        }).start();
                }
                for (int i = 0; i < 10; i++) { // 这行不能改动
                    String input = i + ""; // 这行不能改动
                    try {
                        sq.put(input);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
        // 不能改动此 TestDo 类
        class TestDo {
            public static String doSome(String input) {
                try {
                    Thread. sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                String output = input + ":" + (System.currentTimeMillis() / 1000);
                return output;
            }
        }
```

5.7 链表双向阻塞队列 LinkedBlockingDeque

LinkedBlockingDeque 是一个由链表结构组成的双向阻塞队列。所谓双向队列指的你可以从队列的两端插入和移出元素。双端队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。相比其他的阻塞队列，LinkedBlockingDeque 多了 addFirst、addLast、offerFirst、offerLast、peekFirst、peekLast 等方法，以 First 单词结尾的方法，表示插入、获取（peek）或移除双端队列的第一个元素。以 Last 单词结尾的方法，表示插入，获取或移除双端队列的最后一个元素。另外插入方法 add 等同于 addLast，移除方法 remove 等效于 removeFirst。

在初始化 LinkedBlockingDeque 时可以设置容量防止其过渡膨胀。另外，双向阻塞队列可以运用在“工作窃取”模式中，有点和 LinkedBlockingQueue 类似，这里就不多说了。

5.8 链表传输队列 LinkedTransferQueue

LinkedTransferQueue 是一个由链表结构组成的无界传输阻塞队列。

TransferQueue 是一个继承了 BlockingQueue 的接口，并且增加若干新的方法。LinkedTransferQueue 是实现类，其定义为一个无界的队列，一样具有先进先出（FIFO：first-in-first-out）的特性。Doug Lea 这样评价它：TransferQueue 是一个聪明的队列，它是 ConcurrentLinkedQueue、SynchronousQueue（在公平模式下）、无界的 LinkedBlockingQueues 等的超集。显然易见，混合了若干高级特性，并且具有高性能的一个组合体，一个多面手。单纯从队列来看，TransferQueue 接口增加了一些很实用的新特性。

transfer 算法比较复杂，实现很难看明白。大致的理解是采用所谓双重数据结构（dual data structures）。之所以叫双重，其原因是方法都是通过两个步骤完成：保留与完成。比如，消费者线程从一个队列中取元素，发现队列为空，它就生成一个空元素放入队列，所谓空元素就是数据项字段为空。然后消费者线程在这个字段上继续等待，这叫保留。直到一个生产者线程意欲向队列中放入一个元素，这里它发现最前面的元素的数据项字段为 NULL，它就直接把自己数据填充到这个元素中，即完成了元素的传送。大体是这个意思，这种方式优美地完成了线程之间的高效协作。其 transfer 方法提供了线程之间直接交换对象的捷径的方法，如下所述：

（1）transfer(E e)，若当前存在一个正在等待获取的消费者线程，即立刻移交之；否则，会插入当前元素 e 到队列尾部，并且等待进入阻塞状态，直到有消费者线程取走该元素。

（2）tryTransfer(E e)，若当前存在一个正在等待获取的消费者线程（使用 take()或者 poll()函数），使用该方法会即刻转移/传输对象元素 e；若不存在，则返回 false，并且不进入队列。这是一个不阻塞的操作。

（3）tryTransfer(E e, long timeout, TimeUnit unit)，若当前存在一个正在等待获取的消费者线

程，会立即传输给它；否则将插入元素 e 到队列尾部，并且等待被消费者线程获取消费掉。若在指定的时间内元素 e 无法被消费者线程获取，则返回 false，同时该元素被移除。

（4）hasWaitingConsumer()，很明显，判断是否为终端消费者线程。

（5）getWaitingConsumerCount()，字面意思很明白，获取终端所有等待获取元素的消费线程数量。

（6）size()，因为队列的异步特性，检测当前队列的元素个数需要逐一迭代，可能会得到一个不太准确的结果，尤其是在遍历时有可能队列发生更改。

批量操作类似于 addAll、removeAll、retainAll、containsAll、equals、toArray 等方法，API 不能保证一定会立刻执行。因此，我们在使用过程中，不能有所期待，这是一个具有异步特性的队列。

注意事项：

- 无论是 transfer 还是 tryTransfer 方法，在≥1 个消费者线程等待获取元素时（此时队列为空），都会立刻转交，这属于线程之间的元素交换。注意，这时元素并没有进入队列。
- 在队列中已有数据情况下，transfer 将需要等待前面数据被消费掉，直到传递的元素 e 被消费线程取走为止。
- 使用 transfer 方法，工作者线程可能会被阻塞到生产的元素被消费掉为止。消费者线程等待为零的情况下，各自的处理元素入队与否情况有所不同。
- size()方法，需要迭代，可能不太准确，尽量不要调用。

来看部分 LinkedTransferQueue 的源码，理解一下 LinkedTransferQueue 的实现原理和机制：

```
public class LinkedTransferQueue<E> extends AbstractQueue<E>
  implements TransferQueue<E>, Serializable
{
  public void transfer(E paramE)//转交元素
    throws InterruptedException
  {
    if (xfer(paramE, true, 2, 0L) == null)
      return;
    Thread.interrupted();//线程阻塞
    throw new InterruptedException();
  }

  public boolean tryTransfer(E paramE, long paramLong, TimeUnit paramTimeUnit)
    throws InterruptedException
  {
    if (xfer(paramE, true, 3, paramTimeUnit.toNanos(paramLong)) == null)
      return true;
```

```
        if (!(Thread. interrupted()))
          return false;
        throw new InterruptedException();
      }
    ......
    }
```

LinkedTransferQueue 简单的使用实例如下，来慢慢体会一下：

```
    import java.util.concurrent.TransferQueue;
    public class Consumer implements Runnable {
          private final TransferQueue<String> queue;
          public Consumer(TransferQueue<String> queue) {
                this.queue = queue;
          }
          @Override
          public void run() {
                try {
                      System. out.println(" Consumer " + Thread.currentThread().
getName() + queue.take());
                } catch (InterruptedException e) {
                }
          }
    }
    import java.util.Random;
    import java.util.concurrent.TimeUnit;
    import java.util.concurrent.TransferQueue;
    public class Producer implements Runnable {
          private final TransferQueue<String> queue;
          public Producer(TransferQueue<String> queue) {
                this.queue = queue;
          }
          private String produce() {
                return " your lucky number " + (new Random().nextInt(100));
          }
          public void run() {
                try {
                      while (true ) {
                            if (queue .hasWaitingConsumer()) {
                                  queue.transfer(produce());
```

```
                        }
                        TimeUnit. SECONDS.sleep(1);// 生产者睡眠1s,这样可以看出程序
的执行过程
                    }
                } catch (InterruptedException e) {}
            }
        }
        import java.util.concurrent.LinkedTransferQueue;
        import java.util.concurrent.TransferQueue;
        public class LuckyNumberGenerator {
            public static void main(String[] args) {
                TransferQueue<String> queue = new LinkedTransferQueue<String>();
                Thread producer = new Thread(new Producer(queue));
                producer.setDaemon( true); //设置为守护进程使得线程执行结束后程序自动结束
运行
                producer.start();
                for (int i = 0; i < 10; i++) {
                    Thread consumer = new Thread(new Consumer (queue));
                    consumer.setDaemon( true);
                    consumer.start();
                    try {
                        // 消费者进程休眠1s，以便生产者获得CPU，从而生产产品
                        Thread. sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
```

运行结果如下：

```
Consumer Thread-1 your lucky number 69
Consumer Thread-2 your lucky number 16
Consumer Thread-3 your lucky number 13
Consumer Thread-4 your lucky number 60
Consumer Thread-5 your lucky number 14
Consumer Thread-6 your lucky number 98
Consumer Thread-7 your lucky number 10
Consumer Thread-8 your lucky number 19
```

```
Consumer Thread-9 your lucky number 2
```

总之，BlockingQueue 作为线程容器，可以为线程同步提供有力的保障。BlockingQueue 在线程池里面有重要应用，所以我们这里必须交代清楚。

5.9 同步计数器 CountDownLatch

CountDownLatch 是一个同步辅助类，直译过来就是倒计数（CountDown）门闩（Latch）。倒计数不用说，门闩的意思顾名思义就是阻止前进。在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。用给定的计数初始化 CountDownLatch。由于调用了 countDown() 方法，所以在当前计数到达零之前，await 方法会一直受阻塞。之后，会释放所有等待的线程，await 的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。

主要的方法有：

- CountDownLatch(int count)，构造一个用给定计数初始化的 CountDownLatch。
- void await()，使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断。
- boolean await(long timeout, TimeUnit unit)，使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断或超出了指定的等待时间。
- void countDown()，递减锁存器的计数，如果计数到达零，则释放所有等待的线程。
- long getCount()，返回当前计数。

使用场景：

在一些应用场合中，需要等待某个条件达到要求后才能做后面的事情；同时当线程都完成后也会触发事件，以便进行后面的操作。这个时候就可以使用 CountDownLatch。CountDownLatch 最重要的方法是 countDown()和 await()，前者主要是倒数一次，后者是等待倒数到 0，如果没有到达 0，就只有阻塞等待了。比如，下面两种实际应用场景：

- 应用场景 1：开 5 个多线程去下载，当 5 个线程都执行完了才算下载成功！
- 应用场景 2：当用户多文件上传的时候，可以采用多线程上传，当多个文件都上传成功的时候才算真正的上传成功。

举例：

模拟一个场景，只有三个程序都干完活了，才算项目完成。实例如下：

```
import java.util.concurrent.CountDownLatch;
public class CountDownLatchDemo{
        public static void main(String args[]) throws Exception{
                CountDownLatch latch = new CountDownLatch(3);
```

```
        Worker worker1 = new Worker("Jack 程序员1" ,latch);
        Worker worker2 = new Worker("Rose 程序员2" ,latch);
        Worker worker3 = new Worker("Json 程序员3" ,latch);
        worker1.start();
        worker2.start();
        worker3.start();
        latch.await();
        System. out.println("Main thread end!" );
    }
    static class Worker extends Thread {
        private String workerName ;
        private CountDownLatch latch ;
        public Worker(String workerName,CountDownLatch latch) {
            this.workerName = workerName;
            this.latch = latch;
        }
        @Override
        public void run() {
            try {
                System. out.println("Worker:" +workerName +" is begin.");
                Thread. sleep(1000L);
                System. out.println("Worker:" +workerName +" is end.");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } //模仿干活。
            latch.countDown();
        }
    }
}
```

运行结果如下：

```
Worker:Jack 程序员1 is begin.
Worker:Rose 程序员2 is begin.
Worker:Json 程序员3 is begin.
Worker:Jack 程序员1 is end.
Worker:Json 程序员3 is end.
Worker:Rose 程序员2 is end.
Main thread end!
```

从结果上可以看得出来不像我们之前的实例，MainThread 之后等三个线程同时完成的时候才会继续往下执行。

CountDownLatch 的实现原理：

来看一下 CountDownLatch 的部分源码，理解一下 CountDownLatch 的实现原理和机制。

```
public class CountDownLatch {
    /**
     * CountDownLatch 的核心实现机制利用 AbstractQueuedSynchronizer 简称："AQS"的
state 状态来实现 Count 的阻塞机制。
     */
    private static final class Sync extends AbstractQueuedSynchronizer {
        Sync( int count) {
            setState(count);
        }
        int getCount() {
            return getState();
        }
        protected int tryAcquireShared (int acquires) {
            return (getState() == 0) ? 1 : -1;
        }
        protected boolean tryReleaseShared (int releases) {
            // 覆盖"AQS"的释放状态方法，实现自己的逻辑，来削减 count 线程数
            for (;;) {
                int c = getState();
                if (c == 0)
                    return false ;
                int nextc = c-1;
                if (compareAndSetState(c, nextc))
                    return nextc == 0;
            }
        }
    }
    private final Sync sync ;
    //利用"AQS"的 state 状态，来标示线程的 count 数量
    public CountDownLatch( int count) {
        this.sync = new Sync(count);
    }
    //利用"AQS"获得一个共享模式下的完成状态
    public void await() throws InterruptedException {
```

```
            sync.acquireSharedInterruptibly(1);
        }
        //利用"AQS"获得一个共享模式下的完成状态，超出了指定的等待时间
        public boolean await(long timeout, TimeUnit unit) throws InterruptedException
{
            return sync .tryAcquireSharedNanos(1, unit.toNanos(timeout));
        }
        //调用 AQS 在共享的模式下释放状态也就是数字减一。
        public void countDown() {
            sync.releaseShared(1);
        }
        //调用 AQS 返回当前计数
        public long getCount() {
            return sync .getCount();
        }
    ......
    }
```

其实我们看到 CountDownLatch 源码相对比较简单，主要是利用 AbstractQueuedSynchronizer 来实现。而 AbstractQueuedSynchronizer 其实不难发现，我们如果去查看一下 ReentrantLock、CountDownLatch、Semaphore、FutureTask、ThreadPoolExecutor 的源码的话，都会发现有个名叫 Sync 的静态内部类，继承自 AbstractQueuedSynchronizer。AbstractQueuedSynchronizer 是 java.util.concurrent 的核心组件之一，它为并发包中的其他 synchronizers 提供了一组公共的基础设施。

5.10 抽象队列化同步器 AbstractQueued Synchronizer

AbstractQueuedSynchronizer 是 java.util.concurrent 的核心组件之一，它提供了一个基于 FIFO 队列，可以用于构建锁或者其他相关同步装置的基础框架。该类利用了一个 int 来表示状态，期望它能够成为实现大部分同步需求的基础。使用的方法是继承，子类通过继承同步器并需要实现它的方法来管理其状态，管理的方式就是通过类似 acquire 和 release 的方式来操纵状态。然而，多线程环境中对状态的操纵必须确保原子性，因此子类对于状态的把握，需要使用这个同步器提供的以下三个方法对状态进行操作：

- AbstractQueuedSynchronizer.getState()

- AbstractQueuedSynchronizer.setState(int)
- AbstractQueuedSynchronizer.compareAndSetState(int, int)

子类推荐被定义为自定义同步装置的内部类，就像 CountDownLatch 里面一样，同步器自身没有实现任何同步接口，它仅仅是为定义了若干 acquire 之类的方法提供使用。该同步器既可以作为排他模式也可以作为共享模式，当它被定义为一个排他模式时，其他线程对它的获取就被阻止，而共享模式对于多个线程获取都可以成功。

AbstractQueuedSynchronizer 提供了两种机制：排他模式和共享模式，也可以两种模式共存。处于排他模式时，其他线程试图获取该锁将无法取得成功。在共享模式下，多个线程获取某个锁可能（但不是一定）会获得成功。此类并不“了解”这些不同，除了机械地意识到当在共享模式下成功获取某一锁时，下一个等待线程（如果存在）也必须确定自己是否可以成功获取该锁。处于不同模式下的等待线程可以共享相同的 FIFO 队列。通常，实现子类只支持其中一种模式，但两种模式都可以在（例如）ReadWriteLock 中发挥作用。只支持排他模式或者只支持共享模式的子类不必定义支持未使用模式的方法。

此类通过支持排他模式的子类定义了一个嵌套的 AbstractQueuedSynchronizer.ConditionObject 类，可以将这个类用作 Condition 实现。isHeldExclusively() 方法将报告同步对于当前线程是否是排他的；使用当前 getState() 值调用 release(int) 方法则可以完全释放此对象；如果给定保存的状态值，那么 acquire(int) 方法可以将此对象最终恢复为它以前获取的状态。没有别的 AbstractQueuedSynchronizer 方法创建这样的条件，因此，如果无法满足此约束，则不要使用它。AbstractQueuedSynchronizer.ConditionObject 的行为当然取决于其同步器实现的语义。

此类为内部队列提供了检查、检测和监视方法，还为 condition 对象提供了类似方法。可以根据需要使用，可以使其在同步机制的 AbstractQueuedSynchronizer 的子类中引用这些方法。此类的序列化只存储维护状态的基础原子整数，因此已序列化的对象拥有空的线程队列。需要可序列化的典型子类将定义一个 readObject 方法，该方法在反序列化时将此对象恢复到某个已知初始状态。

接下来看看如何使用：

定义一个子类可以适当地重新定义 tryAcquire(int) 试图在排他模式下获取对象状态，tryRelease(int) 试图设置状态来反映排他模式下的一个释放，tryAcquireShared(int) 试图在共享模式下获取对象状态，tryReleaseShared(int) 试图设置状态来反映共享模式下的一个释放，isHeldExclusively() 表示如果对于当前（正调用的）线程，同步是以排他方式进行的，则返回 true，等等这些方法。这是通过使用 getState()、setState(int) 和/或 compareAndSetState(int, int) 方法来检查和/或修改同步状态来实现的。默认情况下，每个方法都抛出 UnsupportedOperationException。这些方法的实现在内部必须是线程安全的，通常应该很短并且不被阻塞。定义这些方法是使用此类的唯一受支持的方式。其他所有方法都被声明为 final，因为它们无法是各不相同的。AbstractQueuedSynchronizer 内置一个 state 字段，用来表示某种意义——当 ReentrantLock 使用 AQS 的时候，state 被用来表示锁被重入的次数；当 Semaphore 使用 AQS 的时候，state 则被用来表示当前还有多少信号量可被获取。AbstractQueuedSynchronizer 支持的两种模式：排他式和共享式，

两者进行获取和释放动作的思路都是差不多的。

获取同步器的流程如下：

```
if(尝试获取成功){
        return。
}else{
         加入等待队列;park 自己
}
```

释放同步器的流程如下：

```
if(尝试释放成功){
        unpark 等待队列中第一个节点
}else{
        return false
}
```

对概念有了基本的理解之后我们来看一个官方的 demo，就可以实现一个简单的自定义锁了。如下：

```
public class MyLockDemo implements Lock {
        // 内部类，自定义同步器
        private static class Sync extends AbstractQueuedSynchronizer {
                // 是否处于占用状态
                protected boolean isHeldExclusively() {
                        return getState() == 1;
                }
                // 当状态为0的时候获取锁
                public boolean tryAcquire(int acquires) {
                        assert acquires == 1; // Otherwise unused
                        if (compareAndSetState(0, 1)) {
                                setExclusiveOwnerThread(Thread. currentThread());
                                return true ;
                        }
                        return false ;
                }
                // 释放锁，将状态设置为0
                protected boolean tryRelease(int releases) {
                        assert releases == 1; // Otherwise unused
                        if (getState() == 0)
                                throw new IllegalMonitorStateException();
```

```
                setExclusiveOwnerThread( null);
                setState(0);
                 return true ;
            }
            // 返回一个 Condition，每个 condition 都包含了一个 condition 队列
            Condition newCondition() {
                 return new ConditionObject();
            }
        }
        //利用内部类，声明一个 AbstractQueuedSynchronizer 子类，一般都是在内部类里面使用的
        private final Sync sync = new Sync();
        public void lock() {
            sync.acquire(1);
        }
        public boolean tryLock() {
            return sync .tryAcquire(1);
        }
        public void unlock() {
            sync.release(1);
        }
        public boolean isLocked() {
            return sync .isHeldExclusively();
        }
    ......
}
```

最后，AbstractQueuedSynchronizer 子类，一般都是在内部类里面使用的，不同的同步机制和阻塞机制，释放锁和加锁的逻辑可能都不一样，AbstractQueuedSynchronizer 只是提供了一种基于 FIFO 队列的、可以用于构建锁或者其他相关同步装置的基础框架，我们可以结合上面的 CountDownLatch 和下一节将要讲述的 Semaphore 再做一下理解。

5.11 同步计数器 Semaphore

Semaphore 是一个计数信号量。从概念上讲，信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release() 添加一个许可，从而可能释放一个正在阻塞的获取者。就像排队进入上海博物馆一样，先放几个人进去，等这几个人走了，然后再放几个人进入，就像是一种排队机制。

Semaphore 主要的、常用的方法有：

（1）Semaphore(int permits)， 创建具有给定的许可数和给定的非公平的公平设置的 Semaphore 数量。

（2）Semaphore(int permits, boolean fair)，创建具有给定的许可数和给定的公平设置的 Semaphore 数量。

所谓公平性就是是否先进来的先释放，默认是否的。

（3）void acquire()， 从此信号量获取一个许可，在提供一个许可前一直将线程阻塞，否则线程被中断。

（4）int availablePermits()，返回此信号量中当前可用的许可数。

（5）int drainPermits()，获取并返回立即可用的所有许可。

（6）int getQueueLength()，返回正在等待获取的线程的估计数目。

（7）boolean hasQueuedThreads()，查询是否有线程正在等待获取。

（8）boolean isFair()，如果此信号量的公平设置为 true，则返回 true。

（9）protected void reducePermits(int reduction)，根据指定的缩减量减小可用许可的数目。

（10）void release()，释放一个许可，将其返回给信号量。

（11）void release(int permits)，释放给定数目的许可，将其返回到信号量。

（12）boolean tryAcquire()，仅在调用时此信号量存在一个可用许可，才从信号量获取许可。

使用场景：

排队场景，资源有限的房间，资源有限的群等等。常见的实际应用场景包括线程池、连接池等。

实例：

假设一个服务器资源有限，只允许同时 3 个人进行访问，一共来了 10 个人的场景。

```
package demo.thread;
import java.util.concurrent.Semaphore;
public class SemaphoreDemo{
       public static void main(String args[]) throws Exception{
            final Semaphore semaphore = new Semaphore(3);//一次只运行3个人进行访问
            for(int i=0;i<10;i++) {
                  final int no = i;
                 Runnable thread = new Runnable() {
                       public void run (){
                            try {
                                 System. out.println("用户" +no+"连接上了:");
                                 Thread. sleep(300L);
```

```
                                    semaphore.acquire(); //获取接下来执行的许可
                                    System. out.println("用户" +no+"开始访问后台
程序...");
                                    Thread. sleep(1000L);//模仿用户访问服务过程
                                    semaphore.release(); //释放允许下一个线程访问
进入后台
                                    System. out.println("用户" +no+"访问结束。");
                                } catch (InterruptedException e) {
                                    e.printStackTrace();
                                }
                            }
                        };
                         new Thread(thread).start();
                    }
                    System. out.println("Main thread end!" );
                }
            }
```

运行结果如下：

```
用户1连接上了:
用户2连接上了:
用户0连接上了:
用户4连接上了:
用户6连接上了:
Main thread end!
用户8连接上了:
用户3连接上了:
用户7连接上了:
用户9连接上了:
用户5连接上了:
用户0开始访问后台程序...
用户2开始访问后台程序...
用户4开始访问后台程序...
用户0访问结束。
用户6开始访问后台程序...
用户2访问结束。
用户1开始访问后台程序...
用户4访问结束。
用户7开始访问后台程序...
```

```
用户1访问结束。
用户5开始访问后台程序...
用户6访问结束。
用户8开始访问后台程序...
用户7访问结束。
用户3开始访问后台程序...
用户8访问结束。
用户9开始访问后台程序...
用户5访问结束。
用户3访问结束。
用户9访问结束。
```

从结果上可以看的出来，10个人同时进来，但是只能同时3个人访问资源，释放一个允许进来一个。

Semaphore 的实现原理：

来看一下 Semaphore 的部分源码体会一下。

```
public class Semaphore implements java.io.Serializable {
    //实现自己的内部同步类
    private final Sync sync ;
    abstract static class Sync extends AbstractQueuedSynchronizer {
        Sync( int permits) {
            //通过 AbstractQueuedSynchronizer 的状态机制实现信号量的设置
            setState(permits);
        }
        //试图设置状态来反映共享模式下的一个释放，其实就算信号量的值
        final int nonfairTryAcquireShared(int acquires) {
            for (;;) {
                int available = getState();
                int remaining = available - acquires;
                if (remaining < 0 ||
                    compareAndSetState(available, remaining))
                    return remaining;
            }
        }
        //……
    }

    //其实 Semaphore 里面有两种 Sync 来实现
    static final class NonfairSync extends Sync {
        NonfairSync (int permits) {
```

```
            super(permits);
        }
        protected int tryAcquireShared(int acquires) {
            return nonfairTryAcquireShared(acquires);
        }
    }

    //信号量构造函数
    public Semaphore( int permits) {
        sync = new NonfairSync(permits);
    }
    //从此信号量获取一个许可，在提供一个许可前一直将线程阻塞
    public void acquire() throws InterruptedException {
        //调用原生的 AbstractQueuedSynchronizer
        sync.acquireSharedInterruptibly(1);
    }
    // 释放一个许可，将其返回给信号
    public void release() {
        sync.releaseShared(1);
    }
    protected Collection<Thread> getQueuedThreads() {
        return sync .getQueuedThreads();
    }
    //……这里只举例部分代码加以说明
}
```

5.12 同步计数器 CyclicBarrier

CyclicBarrier 是一个同步辅助类，翻译过来叫循环栅栏、循环屏障。它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)，然后所有的这组线程再同步往后面执行。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。不要错误理解了，不是线程屏障可以重复使用，而是多个线程都像多个循环一样，都循环到这个点了，再一起开始往后面执行。

主要的方法有：

（1）CyclicBarrier(int parties)，创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，但它不会在启动 barrier 时执行预定义的操作。

（2）CyclicBarrier(int parties, Runnable barrierAction)，创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，并在启动 barrier 时执行给定的屏障操作，该操作由最后一个进入 barrier 的线程执行。

（3）int await()，在所有参与者都已经在此 barrier 上调用 await 方法之前，将一直等待。

（4）int await(long timeout, TimeUnit unit)，在所有参与者都已经在此屏障上调用 await 方法之前将一直等待，或者超出了指定的等待时间。

（5）int getNumberWaiting()，返回当前在屏障处等待的参与者数目。

（6）int getParties()，返回要求启动此 barrier 的参与者数目。

（7）boolean isBroken()，查询此屏障是否处于损坏状态。

（8）void reset()，将屏障重置为其初始状态。

使用场景：

大数据运算需要拆分成多步骤的时候。比如这么一个实际应用场景：我们需要统计全国的业务数据，其中各省的数据库是独立的，也就是说按省分库，并且统计的数据量很大，统计过程也比较慢。为了提高性能，快速计算，我们采取并发的方式，多个线程同时计算各省数据，每个省下面又用多线程，最后再汇总统计。

实例：

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
public class CyclicBarrierDemo{
        public static void main(String args[]) throws Exception{
                CyclicBarrier barrier = new CyclicBarrier(3,new TotalTask());
                BillTask worker1 = new BillTask("111" ,barrier);
                BillTask worker2 = new BillTask("222" ,barrier);
                BillTask worker3 = new BillTask("333" ,barrier);
                worker1.start();worker2.start();worker3.start();
                System. out.println("Main thread end!" );
        }
        static class TotalTask extends Thread {
                public void run() {
                        System. out.println("所有子任务都执行完了，就开始执行主任务了。" );
                }
        }
        static class BillTask extends Thread {
                private String billName ;
                private CyclicBarrier barrier ;
                public BillTask(String workerName,CyclicBarrier barrier) {
                        this.billName = workerName;this.barrier = barrier;
```

```
            }
            public void run() {
                try {
                    System. out.println("市区:" +billName +"运算开始: " );
                    Thread. sleep(1000L);//模仿第一次运算。
                    System. out.println(" 市区 :" +billName +" 运算完成，等待
中..." );
                     barrier.await();//假设一次运算不完，第二次要依赖第一次的运算
结果。都到达这个节点之后后面才会继续执行。
                    System. out.println("全部都结束，市区" +billName +"才开始后
面的工作。" );
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

运行结果如下：

```
Main thread end!
市区:111运算开始:
市区:333运算开始:
市区:222运算开始:
市区:222运算完成，等待中...
市区:111运算完成，等待中...
市区:333运算完成，等待中...
所有子任务都执行完了，就开始执行主任务了。
全部都结束，市区333才开始后面的工作。
全部都结束，市区222才开始后面的工作。
全部都结束，市区111才开始后面的工作。
```

从运行结果上面仔细体会与 CountDownLatch 的区别。

- CountDownLatch：一个线程（或者多个），等待另外 N 个线程完成某个事情之后才能执行。
- CyclicBarrier：N 个线程相互等待，任何一个线程完成之前，所有的线程都必须等待。

这样比对一下应该就清楚了，对于 CountDownLatch 来说，重点是那个“一个线程”，是它在等待，而另外那 N 的线程在把“某个事情”做完之后可以继续等待，可以终止。而对于 CyclicBarrier 来说，重点是那 N 个线程，它们之间任何一个没有完成，所有的线程都必须等待。

CyclicBarrier 的实现原理是，利用 ReentrantLock 做线程安全锁，实现线程安全等待。来看一下 CyclicBarrier 的部分源码体会一下。

```
public class CyclicBarrier {
    //ReentrantLock 实现线程同步安全
    private final ReentrantLock lock = new ReentrantLock();
    //利用 lock .newCondition 实现主线程的唤醒和等待。
    private final Condition trip = lock .newCondition();
    //parties 表示"必须同时到达 barrier 的线程个数"。
    private final int parties ;
    //parties 个线程到达 barrier 时，会执行的线程
    private final Runnable barrierCommand ;
    //重点看一下 await 方法，发现是调用的 dowait 方法
    public int await() throws InterruptedException, BrokenBarrierException {
        try {
            return dowait(false , 0L);
        } catch (TimeoutException toe) {
            throw new Error(toe); // cannot happen;
        }
    }
    private int dowait( boolean timed, long nanos)
        throws InterruptedException, BrokenBarrierException,
               TimeoutException {
        final ReentrantLock lock = this.lock;
        lock.lock();//加锁
        try {
            final Generation g = generation ;
            if (g.broken )
                throw new BrokenBarrierException();
            if (Thread.interrupted()) {//是否被中断，如果中断，就处理中断逻辑
                breakBarrier();
                throw new InterruptedException();
            }
            int index = --count ;
            //如果 index=0，则意味着"有 parties 个线程到达 barrier"
            if (index == 0) {  // tripped
                boolean ranAction = false;
                try {
                    final Runnable command = barrierCommand ;//都到达线程阀之后要执行
```

的线程

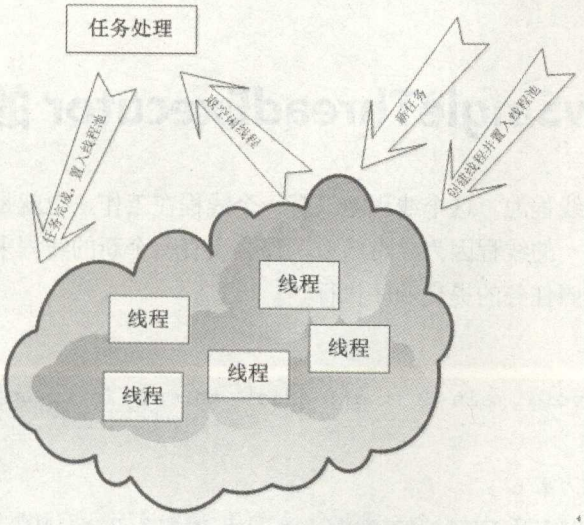
```
                    if (command != null)
                        command.run();
                    ranAction = true;
                    nextGeneration();// 唤醒所有等待线程，并更新 generation
                    return 0;
                } finally {
                    if (!ranAction)
                        breakBarrier();
                }
            }
            // 循环逻辑，直到所有线程数，都到达设置的阀值，或者中断，超时才结束
            for (;;) {
                try {
                    if (!timed)
                        trip.await();
                    else if (nanos > 0L)
                        nanos = trip.awaitNanos(nanos);
                } catch (InterruptedException ie) {
                    if (g == generation && ! g.broken) {
                        breakBarrier();
                        throw ie;
                    } else {
                        //中断线程当发生异常的时候
                        Thread. currentThread().interrupt();
                    }
                }
                if (g.broken )    throw new BrokenBarrierException();
                if (g != generation )  return index;
                // 如果是“超时等待”，并且时间已到，则通过 breakBarrier() 终止
CyclicBarrier，唤醒 CyclicBarrier 中所有等待线程。
                if (timed && nanos <= 0L) {
                    breakBarrier();
                    throw new TimeoutException();
                }
            }
        } finally {
            lock.unlock();//解锁
        }
```

```
        }
    ......
    }
```

总之，这一章重点讲解了线程的队列机制和线程交互等待机制，举例说明了我们平时工作中常用的一些类。作者讲解的内容起到抛砖引玉的作用，给读者一种分析问题的思路和方法，希望我们工作中注意观察和分析，可以体会到多线程编程中的规律，万变不离其宗，掌握好基础，道理都是相通的。

第 6 章
线程池

骐骥一跃，不能十步；驽马十驾，功在不舍。



线程池，特别是高并发项目，互联网项目必须会用的。

6.1 什么是线程池

在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其他更多资源。在 Java 中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗费资源的对象创建和销毁。如何利用已有对象来服务就是一个需要解决的关键问题，其实这就是一些“池化资源”技术产生的原因。比如大家所熟悉的数据库连接池，正是遵循这一思想而产生的。

Java 线程池实现了一个 Java 高并发的、Java 多线程的、可管理的统一调度器。原理和工作机制先不说了，我们对它有个大体的认识，稍后再慢慢学习。先来认识一下

java.util.concurrent.Executors 工作中最常用的和熟知的。

Executors 是个线程的工厂类，方便快速地创建很多线程池，也可以说是一个线程池的工具类。配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是最优的，因此，在 Executors 类里面提供了一些静态工厂，生成一些常用的线程池。常用的方法有以下三种：

- newSingleThreadExecutor：创建一个单线程的线程池。
- newFixedThreadPool：创建固定大小的线程池。
- newCachedThreadPool：创建一个可缓存的线程池。

接下来我们来一一解析。

6.2 newSingleThreadExecutor 的使用

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

先来看一个例子：

```
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
            //使用方法
             ExecutorService executor = Executors.newSingleThreadExecutor();
              for (int i = 0; i < 10; i++) {
                    final int no = i;
                   Runnable runnable = new Runnable() {
                          public void run() {
                                try {
                                    System. out.println("into" + no);
                                    Thread. sleep(1000L);
                                    System. out.println("end" + no);
                                } catch (InterruptedException e) {
                                    e.printStackTrace();
                                }
                          }
                   };
                   executor.execute(runnable);
```

```
            }
            executor.shutdown();
            System. out.println("Thread Main End!" );
    }
```

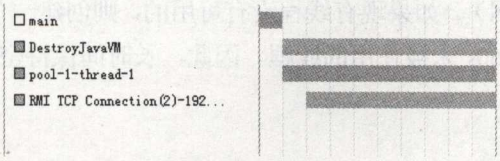
运行结果如下：

```
into0
Thread Main End!
end0
into1
end1
into2
end2
into3
end3
.....
into8
end8
into9
end9
```

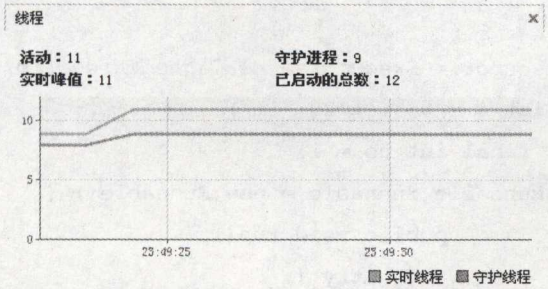
从结果上面看是一条一条地在执行。

从监控工具上面看：

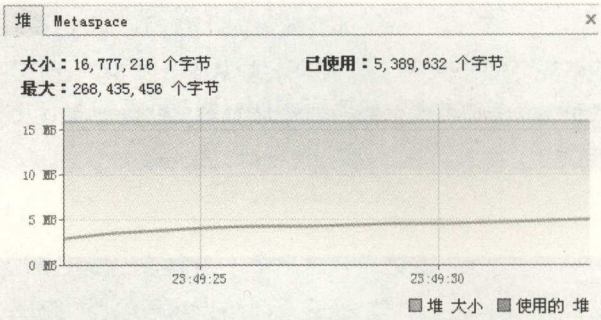
（1）main 线程早就结束了，而线程池里面永远只有一个线程。也可以看出来 10 个线程执行时间也最长，如下图所示。



（2）线程数量永远不变，如下图所示。



（3）使用的 dump 内存基本上处于稳定阶段，如下图所示。



看一下 Executors.newSingleThreadExecutor()的实现方法：

```
public static ExecutorService newSingleThreadExecutor() {
        return new FinalizableDelegatedExecutorService
            ( new ThreadPoolExecutor(1, 1,
                                     0L, TimeUnit. MILLISECONDS,
                                     new LinkedBlockingQueue<Runnable>()));
}
```

上面代码根据 ThreadPoolExecutor 创建一个 LinkedBlockingQueue 的一个大小的线程池，采用默认的异常策略。

6.3 newCachedThreadPool 的使用

创建一个缓存池大小可根据需要伸缩的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60s 未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。

先来看一个例子：

```
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
   ExecutorService executor = Executors. newCachedThreadPool();
              for (int i = 0; i < 20; i++) {
                    final int no = i;
                   Runnable runnable = new Runnable() {
                         public void run() {
                               try {
                                     System. out.println("into" + no);
```

```
                                Thread. sleep(1000L);
                                System. out.println("end" + no);
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        }
                    };
                    executor.execute(runnable);
                }
                executor.shutdown();
                System. out.println("Thread Main End!" );
    }
```

运行结果如下：

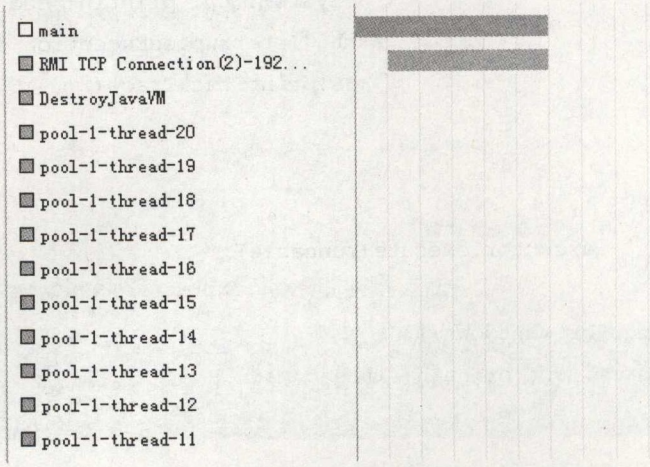
```
into0
into2
into4
into1
into6
......
into15
into18
into19
Thread Main End!
end2
end1
end3
end7
......
end10
end15
end17
end16
end19
```

从结果上面看，一下子所有的线程都开始执行，都在互相的争抢 CPU 资源。

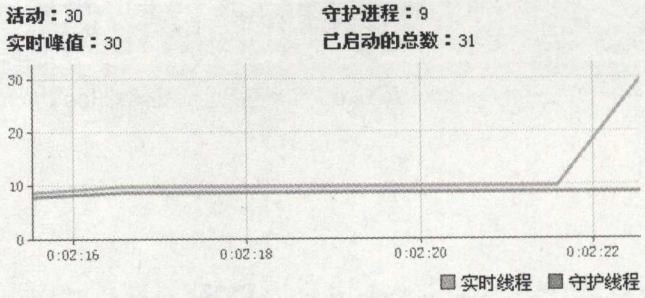
从监控工具种可以看出来：

（1）main 线程一让出资源，线程池里面有 20 个线程同时执行。这时候执行时间也最短，如

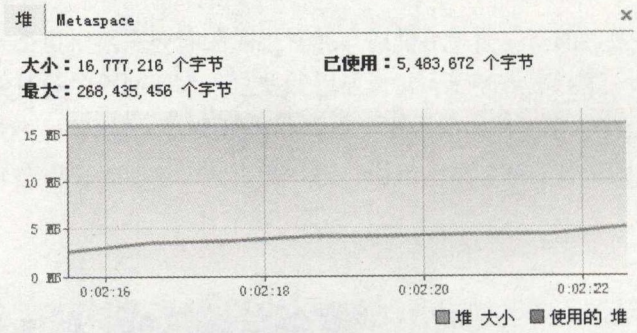
果 CPU 允许执行，也是最快的，如下图所示。



（2）线程数量急剧上升。线程池线程数量没有下降，如下图所示。



（3）使用的 dump 内存基本上处于急剧上升阶段，如下图所示。



看一下 Executors.newCachedThreadPool()的实现：

```
public static ExecutorService newCachedThreadPool() {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit. SECONDS,
                                      new SynchronousQueue<Runnable>());
}
```

代码创建了一个内核线程池。线程为零，来一个线程就在线程池里面创建一个的 SynchronousQueue。

6.4 newFixedThreadPool 的使用

创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。在任意点，在大多数 nThreads 线程会处于处理任务的活动状态。如果在所有线程处于活动状态时提交附加任务，则在有可用线程之前，附加任务将在队列中等待。如果在关闭前的执行期间由于失败而导致任何线程终止，那么一个新线程将代替它执行后续的任务（如果需要）。在某个线程被显式地关闭之前，池中的线程将一直存在。

先来看一个例子：

```
//创建一个固定大小的线程池
ExecutorService executor = Executors. newFixedThreadPool(5);
            for (int i = 0; i < 20; i++) {
                  final int no = i;
                 Runnable runnable = new Runnable() {
                        public void run() {
                              try {
                                    System. out.println("into" + no);
                                    Thread. sleep(1000L);
                                    System. out.println("end" + no);
                              } catch (InterruptedException e) {
                                    e.printStackTrace();
                              }
                        }
                 };
                 executor.execute(runnable);
           }
           executor.shutdown();
           System. out.println("Thread Main End!" );
}
```

运行结果如下：

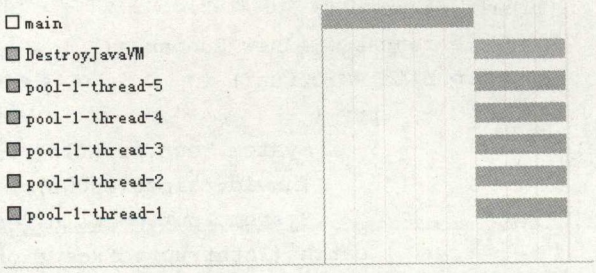
```
into0
into2
Thread Main End!
into4
into1
into3
end0
into5
end2
into6
end4
.....
```

```
end14
into18
end13
into19
end16
end15
end17
end18
end19
```

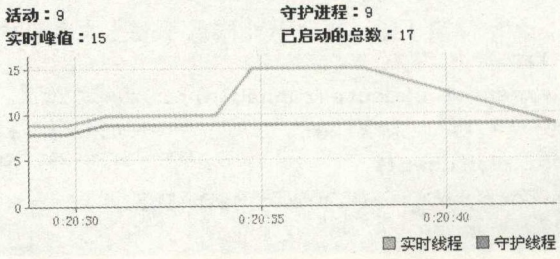
从结果上面看，一下子只有 5 个线程开始执行，然后结束一个再执行一个。

看一下监控工具如下：

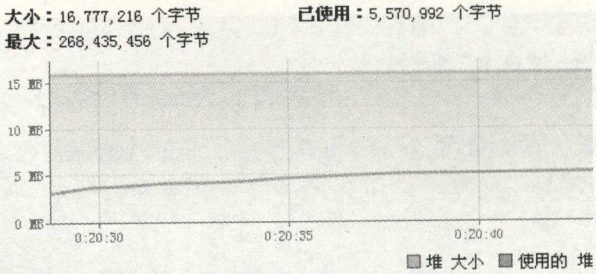
（1）main 线程一让出资源，线程池里面永远有 5 个线程同时执行，如下图所示。这时候执行时间为中等。



（2）线程数量上升到一定数量之后就不变了，然后执行完之后逐渐释放，如下图所示。



（3）使用的 dump 内存上升到线程池的指定大小，基本上处于稳定阶段，如下图所示。



看一下 Executors.newFixedThreadPool() 的实现：

```
public static ExecutorService newFixedThreadPool( int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
```

```
                                    0L, TimeUnit. MILLISECONDS,
                                    new LinkedBlockingQueue<Runnable>());
    }
```

代码创建了一个指定大小的 LinkedBlockingQueue 的线程池。

6.5 线程池的好处

1. 合理利用线程池能够带来 4 个好处

（1）降低资源消耗。通过重复利用已创建的线程，降低线程创建和销毁造成的消耗。

（2）提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

（3）提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配、调优和监控。但是要做到合理地利用线程池，必须对其原理了如指掌。

（4）防止服务器过载，形成内存溢出，或者 CPU 耗尽。

2. 线程池技术如何提高服务器程序的性能

这里所提到服务器程序是指能够接受客户请求并能处理请求的程序，而不只是指那些接受网络客户请求的网络服务器程序。多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。但如果对多线程应用不当，会增加对单个任务的处理时间。

3. 可以举一个简单的例子

假设在一台服务器完成一项任务的时间为 T，并假设：

- T1，创建线程的时间。
- T2，在线程中执行任务的时间，包括线程间同步所需时间。
- T3，线程销毁的时间。

显然 T ＝ T1＋T2＋T3。注意这是一个极度简化的假设。

可以看出 T1、T3 是多线程本身的带来的开销，我们渴望减少 T1、T3 所用的时间，从而减少 T 的时间。但一些线程的使用者并没有注意到这一点，所以在程序中频繁地创建或销毁线程，这导致 T1 和 T3 在 T 中占有相当比例。显然这是突出了线程的弱点（T1，T3），而不是优点（并发性）。

线程池技术正是关注如何缩短或调整 T1、T3 时间的技术，从而提高服务器程序性能的。它把 T1、T3 分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段，这样在服务器程序处理客户请求时，不会有 T1、T3 的开销了。

线程池不仅调整 T1、T3 产生的时间段，而且它还显著减少了创建线程的数目。再看一个例子：

假设一个服务器一天要处理 50 000 个请求，并且每个请求需要一个单独的线程完成。我们比较一下利用线程池技术和不利用线程池技术的服务器处理这些请求时所产生的线程总数。在线程池中，线程数一般是固定的，所以产生线程总数不会超过线程池中线程的数目或者上限（以下简称线程池尺寸），而如果服务器不利用线程池来处理这些请求则线程总数为 50 000。一般线程池尺寸是远小于 50 000。所以利用线程池的服务器程序不会为了创建 50 000 而在处理请求时浪费时间，从而提高效率。

4. 线程池的应用范围

（1）需要大量的线程来完成任务，且完成任务的时间比较短。Web 服务器完成网页请求这样的任务，使用线程池技术是非常合适的。因为单个任务小，而任务数量巨大，你可以想象一个热门网站的点击次数。但对于长时间的任务，比如一个 Telnet 连接请求，线程池的优点就不明显了。因为 Telnet 会话时间比线程的创建时间大多了。

（2）对性能要求苛刻的应用，比如要求服务器迅速响应客户请求。

（3）接受突发性的大量请求，但不至于使服务器因此产生大量线程的应用。突发性大量客户请求，在没有线程池情况下，将产生大量线程，虽然理论上大部分操作系统线程数目最大值不是问题，短时间内产生大量线程可能使内存到达极限，并出现 OutOfMemory 的错误。

6.6 线程池的工作机制及其原理

线程池的核心的两个队列：

- 线程等待池，即线程队列 BlockingQueue。
- 任务处理池 (PoolWorker)，即正在工作的 Thread 列表（HashSet<Worker>）。

线程池的核心的参数：

- 核心池大小（corePoolSize），即固定大小，设定好之后，线程池的稳定峰值，达到这个值之后池的线程数大小不会释放的。
- 最大处理线程池数（maximumPoolSize），当线程池里面的线程数超过 corePoolSize，小于 maximumPoolSize 时会动态创建与回收线程池里面的线程的资源。

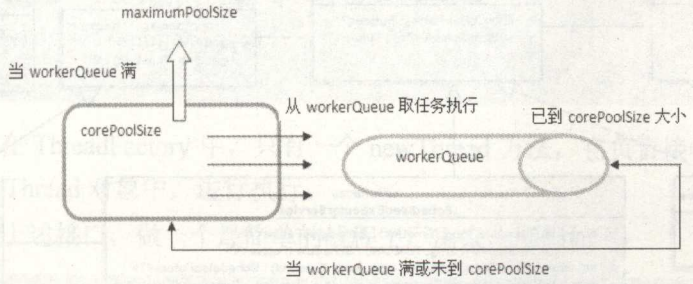
线程池的运行机制：

我们举一个例子来说明。假如有一个工厂，工厂里面有 10 个工人，每个工人同时只能做一件任务。因此只要当 10 个工人中有工人是空闲的，来了任务就分配给空闲的工人做；当 10 个工人都有任务在做时，如果还来了任务，就把任务进行排队等待。

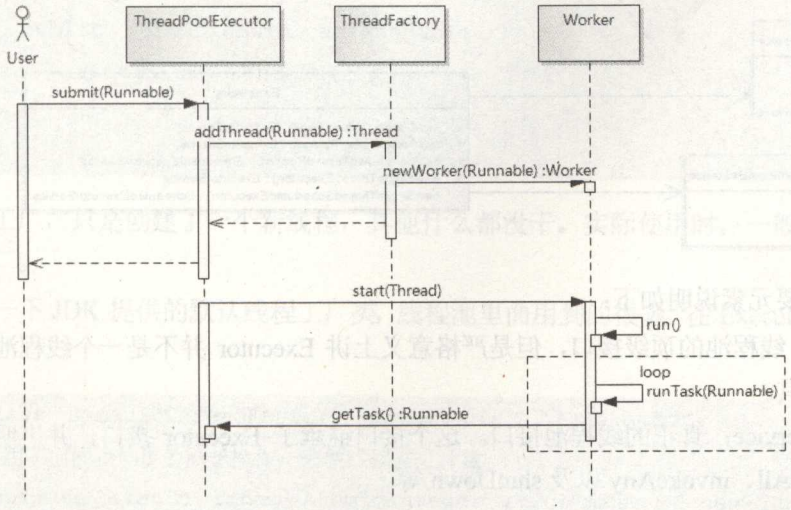
如果说新任务数目增长的速度远远大于工人做任务的速度，那么此时工厂主管可能会想补救措施，比如重新招 4 个临时工人进来；然后就将任务也分配给这 4 个临时工人做。

如果说这 14 个工人做任务的速度还是不够，此时工厂主管可能就要考虑不再接收新的任务或者抛弃前面的一些任务了。当这 14 个工人当中有人空闲时，而新任务增长的速度又比较缓慢，工厂主管可能就考虑辞掉 4 个临时工了，只保持原来的 10 个工人，毕竟请额外的工人是要花钱的。

这个例子中永远等待干活的 10 个工人机制就是 workerQueue。这个例子中的 corePoolSize 就是 10，而 maximumPoolSize 就是 14（10+4）。也就是说 corePoolSize 就是线程池大小，maximumPoolSize 在我看来是线程池的一种补救措施，即任务量突然过大时的一种补救措施。再看看下图好好理解一下。工人永远在等待干活，就像 workerQueue 永远在循环干活一样，除非，整个线程池停止了。



线程池里面的线程的时序图如下图所示：

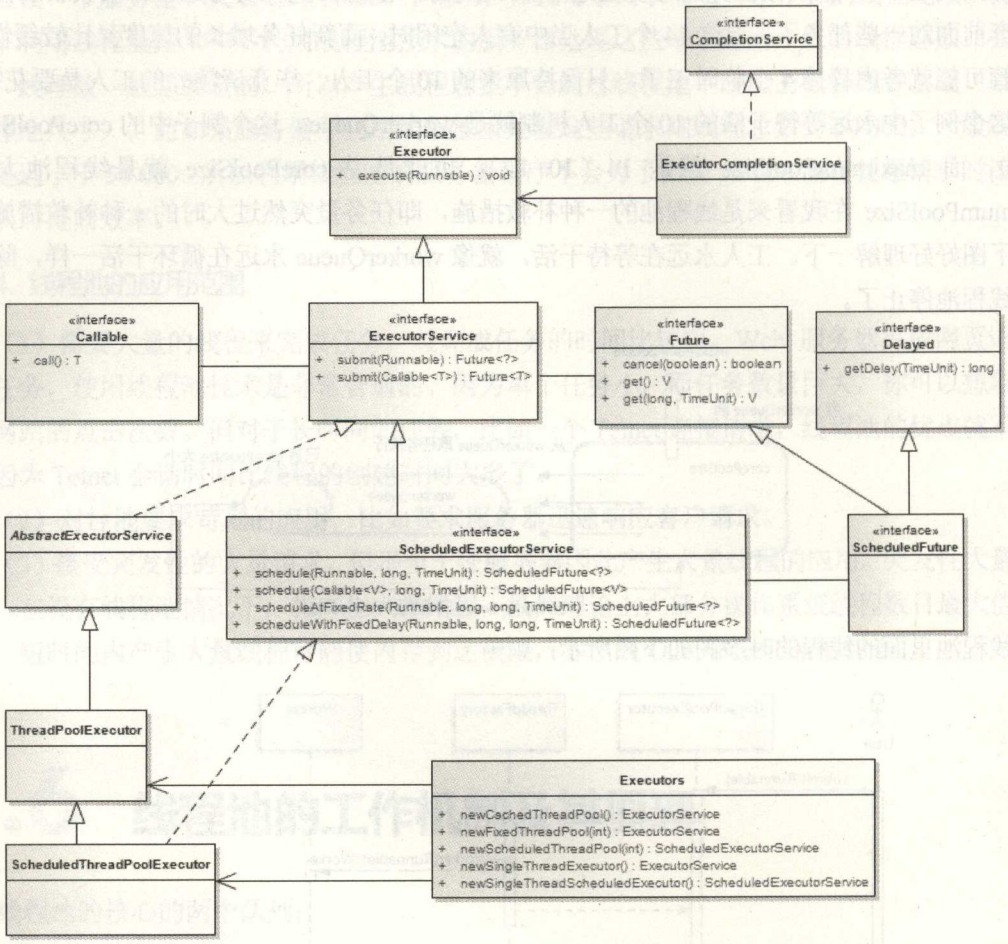


6.7 自定义线程池与 ExecutorService

自定义线程池需要用到 ThreadFactory，本节将通过创建一个线程的例子对 ExecutorService 及其参数进行详细讲解。

1. 认识一下 ExecutorService 的家族

ExecutorService 类的家族成员如下图所示。



上图中主要元素说明如下：

Executor：线程池的顶级接口，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。

ExecutorService：真正的线程池接口。这个接口继承了 Executor 接口，并声明了一些方法：submit、invokeAll、invokeAny 以及 shutDown 等。

AbstractExecutorService 实现了 ExecutorService 接口，基本实现了 ExecutorService 中声明的所有方法。

ThreadPoolExecutor：ExecutorService 的默认实现，继承了类 AbstractExecutorService 。

ScheduledExecutorService：与 Timer/TimerTask 类似，解决那些需要任务重复执行的问题。

ScheduledThreadPoolExecutor：继承 ThreadPoolExecutor 的 ScheduledExecutorService 接口实现，周期性任务调度的类实现。

Executors 是个线程的工厂类，方便快速创建很多线程池。

2. 利用 ThreadFactory 创建一个线程

java.util.concurrent.ThreadFactory 提供了一个创建线程工厂的接口。工厂模式是我们学习编程时，接触到的第一个设计模式，也是最简单、最常用的一个设计模式。在 JDK 的源码中，大量使用工厂模式，ThreadFactory 就是其中一种。

前面我们介绍了三种创建线程的方法，这里我们再介绍一种，通过线程工厂直接创建线程。设想这样一种场景，我们需要一个线程池，并且对于线程池中的线程对象，赋予统一的线程优先级、统一的名称、甚至进行统一的业务处理或/和业务方面的初始化工作，这时工厂方法就是最好用的方法了。

ThreadFactory 的接口内容如下：

```
public interface ThreadFactory {
    Thread newThread(Runnable r);
}
```

我们可以看到在 ThreadFactory 中，只有一个 newThread 方法，它负责接收一个 Runnable 对象，并将其封装到 Thread 对象中，进行执行。

我们可以实现上述接口，做一个最简单的线程工厂出来，源码如下：

```
public class SimpleThreadFactory implements ThreadFactory{
        @Override
        public Thread newThread(Runnable r) {
            return new Thread(r);
        }
}
```

上述线程工厂，只是创建了一个新线程，其他什么都没干。实际使用时，一般不会创建这么简单的线程工厂。

我们来看一下 JDK 提供的默认线程工厂类，线程池里面用到的很多。在 Executors 工具类中，代码如下：

```
static class DefaultThreadFactory implements ThreadFactory {
        //用了前面我们讲的原子操作，来做线程安全计数
        private static final AtomicInteger poolNumber = new AtomicInteger(
1);

        private final ThreadGroup group;
        private final AtomicInteger threadNumber = new AtomicInteger(1);
        private final String namePrefix;

        DefaultThreadFactory() {
            SecurityManager s = System. getSecurityManager();
```

```
            group = (s != null) ? s.getThreadGroup() :
                                  Thread.currentThread().getThreadGroup();
            namePrefix = "pool-" +
                         poolNumber.getAndIncrement() +
                        "-thread-";
        }

        public Thread newThread(Runnable r) {
            Thread t = new Thread(group , r,
                                  namePrefix + threadNumber.getAndIncrement(),
                                  0);
            if (t.isDaemon())
                t.setDaemon( false);
            if (t.getPriority() != Thread.NORM_PRIORITY)
                t.setPriority(Thread. NORM_PRIORITY);
            return t;
        }
    }
```

所以我们看到线程池里面的线程的名称，都是这里来的哦，如果想自定义，扩展即可。

3. 理解 RejectedExecutionHandler

如果线程池的线程已经饱和，并且任务队列也已满,那么就需要做丢弃处理，RejectedExecutionHandler 这个类就是用来处理被丢弃的线程的异常处理接口。

接口内容如下：

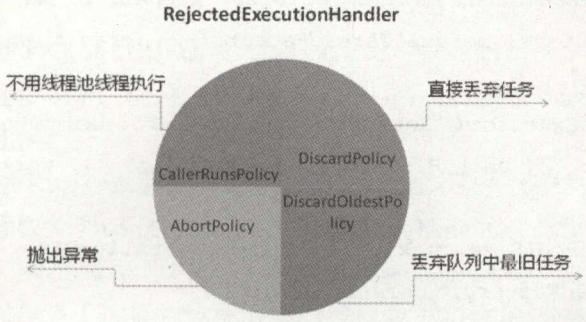
```
public interface RejectedExecutionHandler {
    //被线程池丢弃的线程处理机制
    void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
}
```

可以自己实现这个接口实现自己的线程丢弃处理类，简单的示例代码如下：

```
public class RejectedExecutionHandlerDemo implements RejectedExecutionHandler {
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        System. out.println("线程信息:" + r.toString() +" 被遗弃的线程池:
" +e.toString());
    }
}
```

JDK 里面 RejectedExecutionHandler 提供了 4 种方式来处理任务拒绝策略，如下图所示。

线程池任务拒绝策略



- AbortPolicy：直接抛出异常。
- CallerRunsPolicy：只用调用者所在线程来运行任务。
- DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
- DiscardPolicy：不处理，丢弃掉。

大家去看几个类的源码也非常简单，现在举一种 AbortPolicy 来简单说明一下。

```
public static class AbortPolicy implements RejectedExecutionHandler {
        public AbortPolicy() { }
        //直接抛出异常
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            throw new RejectedExecutionException( "Task " + r.toString() +
                                                  " rejected from " +
                                                  e.toString());
        }
}
```

为什么有任务拒绝的情况发生：

这里先假设一个前提：线程池有一个任务队列，用于缓存所有待处理的任务，正在处理的任务将从任务队列中移除。因此，在任务队列长度有限的情况下，就会出现新任务的拒绝处理问题，需要有一种策略来处理这种应该加入任务队列却因为队列已满无法加入的情况。另外，在线程池关闭的时候，也需要对任务加入队列操作进行额外的协调处理。

4. ThreadPoolExecutor 详解

ThreadPoolExecutor 类是线程池中最核心的一个类，因此如果要透彻地了解 Java 中的线程池，必须先了解这个类。下面我们来看一下 ThreadPoolExecutor 类的具体实现源码。

4 种构造方法：

```
    public ThreadPoolExecutor (int corePoolSize,int maximumPoolSize,long keepAli
veTime,TimeUnit unit,BlockingQueue<Runnable> workQueue) {
            this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
                Executors. defaultThreadFactory(), defaultHandler);
    }
    public ThreadPoolExecutor( int corePoolSize,int maximumPoolSize,long keepAl
iveTime,     TimeUnit     unit,BlockingQueue<Runnable>     workQueue,ThreadFactory
threadFactory) {
            this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
                threadFactory, defaultHandler);
    }
    public ThreadPoolExecutor( int corePoolSize,int maximumPoolSize,long keepAl
iveTime,TimeUnit  unit,BlockingQueue<Runnable>  workQueue,RejectedExecutionHandler
handler) {
            this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
                Executors. defaultThreadFactory(), handler);
        }
    Public ThreadPoolExecutor (int corePoolSize,int maximumPoolSize,long keepAli
veTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,ThreadFactory
threadFactory,RejectedExecutionHandler handler) {
            if (corePoolSize < 0 ||maximumPoolSize <= 0 ||maximumPoolSize <
corePoolSize ||keepAliveTime < 0)
                throw new IllegalArgumentException();
            if (workQueue == null || threadFactory == null || handler == null)
                throw new NullPointerException();
            this.corePoolSize = corePoolSize;
            this.maximumPoolSize = maximumPoolSize;
            this.workQueue = workQueue;
            this.keepAliveTime = unit.toNanos(keepAliveTime);
            this.threadFactory = threadFactory;
            this.handler = handler;
    }
```

通过观察每个构造器的源码的具体实现，发现前面三个构造器都是调用第 4 个构造器进行的初始化工作。也可以说这是一个真正的构造方法，我们详细说明一下里面的参数：

（1）int corePoolSize：核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了 prestartAllCoreThreads()或者 prestartCoreThread()方法，从这 2 个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建 corePoolSize 个线程

或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为 0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到 corePoolSize 后，就会把到达的任务放到缓存队列当中。

（2）int maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；在 corePoolSize 和 maximumPoolSize 的线程数会被自动释放。而小于 corePoolSize 的不会。

（3）long keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于 corePoolSize 时，keepAliveTime 才会起作用，直到线程池中的线程数不大于 corePoolSize，即当线程池中的线程数大于 corePoolSize 时，如果一个线程空闲的时间达到 keepAliveTime，则会终止，直到线程池中的线程数不超过 corePoolSize。但是如果调用了 allowCoreThreadTimeOut(boolean) 方法，在线程池中的线程数不大于 corePoolSize 时，keepAliveTime 参数也会起作用，直到线程池中的线程数为 0。

（4）TimeUnit unit：参数 keepAliveTime 的时间单位，一个 JDK 里面的时间单位枚举类。

（5）BlockingQueue workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列就是我们前面讲过的几种选择(ArrayBlockingQueue;LinkedBlockingQueue;SynchronousQueue;)。

（6）ThreadFactory threadFactory：线程工厂，主要用来创建线程；可以是一个自定义的线程工厂，默认就是我们前面讲的 Executors.defaultThreadFactory()。用来在线程池里面创建线程。

（7）RejectedExecutionHandler handler：表示当拒绝处理任务时的策略，也是可以自定义的，默认也是我们前面说过的 4 种取值：

- ThreadPoolExecutor.AbortPolicy（默认的）
- ThreadPoolExecutor.DiscardPolicy
- ThreadPoolExecutor.DiscardOldestPolicy
- ThreadPoolExecutor.CallerRunsPolicy

所以想自定义线程池就从以上几个参数入手。接下来我们看 ThreadPoolExecutor 里面的一些具体源码，稍微理解一下里面的实现原理：

```
    //默认异常处理机制
    private static final RejectedExecutionHandler defaultHandler = new AbortPolicy(
);
    //任务缓存队列，用来存放等待执行的任务
    private final BlockingQueue<Runnable> workQueue;
    //线程池的主要状态锁，对线程池状态（比如线程池大小、runState 等）的改变都要使用这个锁
    private final ReentrantLock mainLock = new ReentrantLock();
    private final HashSet<Worker> workers = new HashSet<Worker>();  //用来存放工作集
    //volatile 可变变量关键字，写的时候用 mainLock 做锁，读的时候无锁，高性能的。
    private volatile long  keepAliveTime;    //线程存货时间
```

```
private volatile boolean allowCoreThreadTimeOut; //是否允许为核心线程设置存活时间
//核心池的大小（即线程池中的线程数目大于这个参数时，提交的任务会被放进任务缓存队列）
private volatile int    corePoolSize;
private volatile int    maximumPoolSize;    //线程池最大能容忍的线程数
private volatile int    poolSize;           //线程池中当前的线程数
private volatile RejectedExecutionHandler handler; //任务拒绝策略
```

结合我们以前讲的原理我相信通过以上的几个参数 大概就能猜的出来里面是怎么实现的了。

5. 自定义实现一个简单的 Web 请求线程池

我们来自定义实现一个简单的 Web 请求线程池。模仿 Web 服务的需求场景说明如下：

- 服务器可容纳的最小请求数是多少。
- 可以动态扩充的请求数大小是多少。
- 多久回收多余线程数即请求数。
- 用户访问量大了怎么处理。
- 线程队列机制采取有优先级的排队的执行机制。

根据上面定义的场景，我们来看看这个自定义线程池该如何写？代码如下：

```
public class MyExecutors extends Executors{
    //利用默认线程工厂和 PriorityBlockingQueue 队列机制，当然了，我们也可以分别实现工厂类和
继承 queue 进行自定和扩展
    public static ExecutorService
newMyWebThreadPool(int minSpareThreads,int maxThreads,int maxIdleTime) {
        return new ThreadPoolExecutor(minSpareThreads, maxThreads, maxIdleTime,
TimeUnit.MILLISECONDS , new PriorityBlockingQueue<Runnable>());
    }
}
```

当然了我们可能只是简单地做了一些自定义的实现，大家可以仔细体会一下，后面我们会详细说明一下真正的 Tomcat 里面的线程池是如何实现的。

6.8 线程池在工作中的错误使用

（1）分不清楚线程池是单例还是多对象。先问各位同学一个问题，线程池在使用的过程中是单例的还是多例的？

线程池一定要在合理的单例模式下才有效，工作中我发现有些同学将线程池的创建方法放在

services 方法里面去创建线程池，这是不可以的，因为每当这个方法被调用的时候不是创建多少个线程的问题了，而是创建出来了一大堆线程池！

（2）线程池数量设置很大，请求过载。这样就发挥不了线程池的另外一个优点了。

仅仅是请求就压垮了服务器，这种情况是可能的。在这种情形下，我们可能不想将每个到来的请求都到我们的工作队列中排队，因为排在队列中等待执行的任务可能会消耗太多的系统资源并引起资源缺乏。在这种情形下决定如何做取决于你自己；在某些情况下，你可以简单地抛弃请求，依靠更高级别的协议稍后重试请求，你也可以用一个指出服务器暂时很忙的响应来拒绝请求。

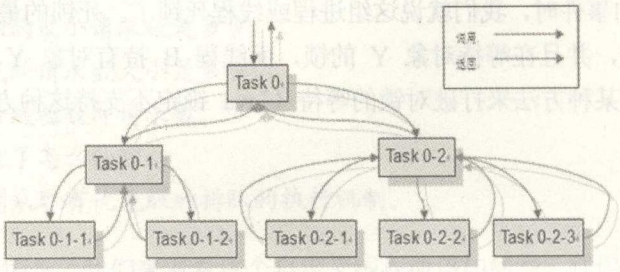
（3）注意死锁问题。

任何多线程应用程序都有死锁风险。当一组进程或线程中的每一个都在等待一个只有该组中另一个进程才能引起的事件时，我们就说这组进程或线程死锁了。死锁的最简单情形是：线程 A 持有对象 X 的独占锁，并且在等待对象 Y 的锁，而线程 B 持有对象 Y 的独占锁，却在等待对象 X 的锁。除非有某种方法来打破对锁的等待（Java 锁定不支持这种方法），否则死锁的线程将永远等下去。

第 7 章

JDK7新增的Fork/Join

卓越是方向，成就在路上。



化繁为简，分而治之。递归的分解和合并，直到任务小到可以接受的程度。

7.1 认识 Future 任务机制和 FutureTask

本节我们介绍 Future 类。前面我们提到了 thread 的三种创建方式，一种是返回结果的就是要实现 Callable 接口。下面我们来看一下这个接口源码：

```
public interface Callable <V> {
    //只有一个获得返回结果的方法，实现这个方法即可。
    V call() throws Exception;
}
```

Future 类就是对于具体的 Runnable 或者 Callable 任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过 get 方法获取执行结果，该方法会阻塞直到任务返回结果。Future 类位于 java.util.concurrent 包下，它也是一个接口，如下：

```
public interface Future<V> {
    //用来取消任务，如果取消任务成功则返回 true，如果取消任务失败则返回 false。参数 mayInterruptIfRunning 表示是否允许取消正在执行却没有执行完毕的任务，如果设置 true，则表示可以取消正在执行过程中的任务。如果任务已经完成，则无论 mayInterruptIfRunning 为 true 还是 false，此方
```

法肯定返回 false，即如果取消已经完成的任务会返回 false；如果任务正在执行，若 mayInterruptIfRunning 设置为 true，则返回 true，若 mayInterruptIfRunning 设置为 false，则返回 false；如果任务还没有执行，则无论 mayInterruptIfRunning 为 true 还是 false，肯定返回 true。

```
    boolean cancel( boolean mayInterruptIfRunning);
    //表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。
    boolean isCancelled();

    //表示任务是否已经完成，若任务完成，则返回 true。
    boolean isDone();

    //用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回。
    V get() throws InterruptedException, ExecutionException;

    //用来获取执行结果，如果在指定时间内，还没获取到结果，就直接返回 null
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

也就是说 Future 提供了三种功能：

- 判断任务是否完成。
- 能够中断任务。
- 能够获取任务执行结果。

因为 Future 只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的 FutureTask。FutureTask 目前是 Future 接口的一个唯一实现类。

我们先简单地来看一下 FutureTask 类：

```
public class FutureTask<V> implements RunnableFuture<V>
```

FutureTask 类实现了 RunnableFuture 接口，我们看一下 RunnableFuture 接口的实现：

```
public interface RunnableFuture <V> extends Runnable, Future<V> {
    void run();
}
```

可以看出 RunnableFuture 继承了 Runnable 接口和 Future 接口，而 FutureTask 实现了 RunnableFuture 接口。所以它既可以作为 Runnable 被线程执行，又可以作为 Future 得到 Callable 的返回值。

FutureTask 提供了 2 个构造器：

```
public FutureTask(Callable<V> callable) {
```

```
        //创建一个 FutureTask，一旦运行就执行给定的 Callable
    }
    public FutureTask(Runnable runnable, V result){
        //创建一个 FutureTask，一旦运行就执行给定的 Runnable，并安排成功完成时 get 返回给定的
结果 。
    }
```

使用场景：

实际工作中，可能需要统计各种类型的报表呈现结果，可能一个大的报表要依赖很多小的模块的运算结果，一个线程做可能又比较慢，就可以拆分成 N 多个小线程，然后将其结果合并起来作为大的报表呈现结果。而接下来的 Fork/Join 就是基于 Future 实现的。

看下面的实例先来体验一下：

```
package demo.thread;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;
public class FutureTaskDemo {
        public static void main(String[]        args) throws InterruptedException,
ExecutionException {
                SonTask task1 = new SonTask("Thread Son1" );
                FutureTask<String> f1 = new FutureTask<String>(task1);
                 new Thread(f1).start();
                System. out.println(f1.get());//只有得到返回结果后才会继续往下面执行
                FutureTask<Integer> f2 = new FutureTask<Integer>(new MyRun(), 22);//
执行完指定线程，返回指定结果
               new Thread(f2 ).start();
                System. out.println("result_" +f2.get());//只有得到指定结果后才会继续往
下面执行
        }
}
class SonTask implements Callable<String> {
        private String name = "";
        SonTask(String name) {
                this.name = name;
        }
        @Override
        public String call() throws Exception {
                Thread. sleep(1000L);
```

```
            System. out.println(name +"任务计算完成" );
             return "result_11" ;
        }
    }
    class MyRun implements Runnable {
        @Override
        public void run() {
             try {
                  Thread. sleep(1000L);//模拟干活
             } catch (InterruptedException e) {
                  e.printStackTrace();
             }
            System. out.println("特定线程2完成" );
        }
    }
```

运行结果如下：

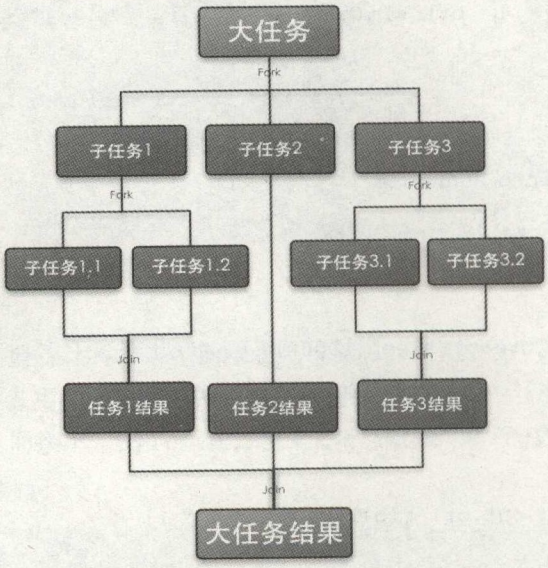
```
Thread Son1任务计算完成
result_11
特定线程2完成
result_22
Thread Main End!
```

从结果上可以看到是按照预想结果，按顺序，按步骤地执行。

7.2 什么是 Fork/Join 框架

Fork/Join 框架是 Java 7 提供的一个用于并行执行任务的框架， 是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

我们再通过 Fork 和 Join 这两个单词来理解下 Fork/Join 框架，Fork 就是把一个大任务切分为若干子任务并行地执行，Join 就是合并这些子任务的执行结果，最后得到这个大任务的结果。比如计算 1+2+……+10000，可以分割成 10 个子任务，每个子任务分别对 1 000 个数进行求和，最终汇总这 10 个子任务的结果。Fork/Join 的运行流程图如下：



让我们通过一个简单的需求来体验一下什么是 Fork / Join 框架，需求是：计算 1+2+3+4 的结果。

使用 Fork / Join 框架首先要考虑到的是如何分割任务。如果我们希望每个子任务最多执行两个数的相加，那么我们设置分割的阈值是 2，由于是 4 个数字相加，所以 Fork / Join 框架会把这个任务 fork 成两个子任务，子任务一负责计算 1+2，子任务二负责计算 3+4，然后再 join 两个子任务的结果。

先来看一个例子体会一下如下：

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
import java.util.concurrent.RecursiveTask;
public class ForkJoinTaskDemo {
        public static void main(String[]        args) throws InterruptedException,
ExecutionException{
                ForkJoinPool forkJoinPool = new ForkJoinPool();
                CountTask task = new CountTask(1,5);
                Future<Integer> result = forkJoinPool.submit(task);
                System. out.println("1-5最终相加的结果: " + result.get());
                CountTask task2 = new CountTask(1,100);
                Future<Integer> result2 = forkJoinPool.submit(task2);
                System. out.println("1-100最终相加的结果: " + result2.get());
                System. out.println("Thread Main End!" );
        }
}
```

```
class CountTask extends RecursiveTask<Integer> {
    private static final long serialVersionUID = 3336021432713606929L;
    private static int splitSize =2;
    private int start,end ;
    public CountTask(int start,int end) {
        this.start = start;
        this.end = end;
    }
    @Override
    protected Integer compute() {
        int sum=0;
        //如果任务已经不需要再拆分了就开始计算
        boolean canCompute=(end -start )<=splitSize;
        if(canCompute) {
            for(int i=start;i<= end;i++) {
                sum=sum+i;
            }
        } else {
            //拆分成两个子任务
            int middle = (start +end )/2;
            CountTask fistTask = new CountTask(start ,middle);
            CountTask secondTask = new CountTask(middle+1,end );
            fistTask.fork(); //开始执行
            secondTask.fork(); //
            //获得第一个子任务的结果，得不到结果，此线程不会往下面执行。
            int firstResult = fistTask.join();
            int secondResult = secondTask.join();
            //合并两个儿子的执行结果。
            sum = firstResult+ secondResult;
        }
        return sum;
    }
}
```

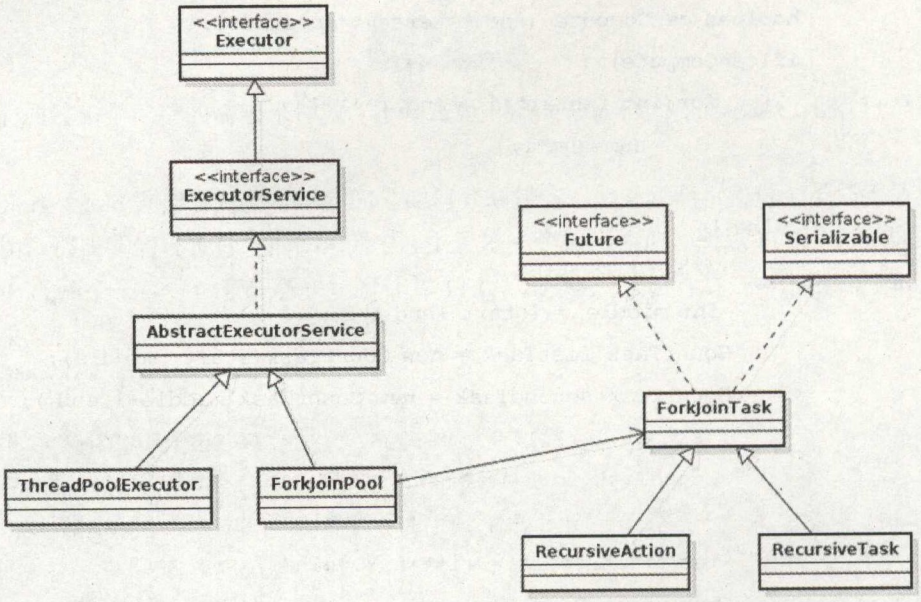
运行结果如下：

```
1-5最终相加的结果：15
1-100最终相加的结果：5050
Thread Main End!
```

通过这个例子让我们再来进一步了解一下 ForkJoinTask，ForkJoinTask 与一般的任务的主要区别在于它需要实现 compute 方法，在这个方法里，首先需要判断任务是否足够小，如果足够小就直接执行任务。如果不足够小，就必须分割成两个子任务，每个子任务在调用 fork 方法时，又会进入 compute 方法，看看当前子任务是否需要继续分割成孙任务，如果不需要继续分割，则执行当前子任务并返回结果。使用 join 方法会等待子任务执行完成并得到其结果。

7.3 认识 Fork/Join 的 JDK 里面的家族

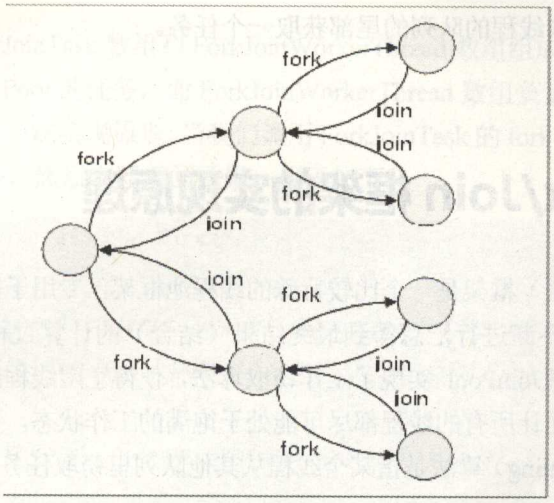
本节我们学习一下 Fork/Join 家族成员，如下图所示。



先来看一下 ForkJoinTask，它是实现 Future 的另一种有返回结果的实现方法，比 Future 多了两个重要的方法：

- fork ()，这个方法决定了 ForkJoinTask 的异步执行，凭借这个方法可以创建新的任务。
- join ()，该方法负责在计算完成后返回结果，因此允许一个任务等待另一任务执行完成。

Fork /join 的完整过程如下图所示：



- RecursiveAction：继承 ForkJoinTask，用于没有返回结果的任务。
- RecursiveTask：继承 ForkJoinTask，用于有返回结果的任务。
- ForkJoinPool：和线程池 ThreadPoolExecutor 一样都是实现的 Executor 接口。

ForkJoinPool 提供了三个方法来调度子任务：

- execute 异步执行指定的任务。
- invoke 和 invokeAll 执行指定的任务，等待完成，返回结果。
- submit 异步执行指定的任务，并立即返回一个 Future 对象。

我们已经很清楚 Fork/Join 框架的需求了，那么我们可以思考一下，如果让我们来设计一个 Fork/Join 框架，该如何设计？这个思考有助于你理解 Fork/Join 框架的设计。

第一步分割任务。首先我们需要有一个 fork 类来把大任务分割成子任务，有可能子任务还是很大，所以还需要不停地分割，直到分割出的子任务足够小。

第二步执行任务并合并结果。分割的子任务分别放在双端队列里，然后几个启动线程分别从双端队列里获取任务执行。子任务执行完的结果都统一放在一个队列里，启动一个线程从队列里拿数据，然后合并这些数据。

Fork/Join 使用两个类来完成以上两件事情：

（1）ForkJoinTask：我们如果要使用 ForkJoin 框架，必须首先创建一个 ForkJoin 任务。它提供在任务中执行 fork() 和 join() 操作的机制，通常情况下我们不需要直接继承 ForkJoinTask 类，而只需要继承它的子类，重载 protected void compute() 方法。Fork/Join 框架提供了以下两个子类：

- RecursiveAction：用于没有返回结果的任务。
- RecursiveTask ：用于有返回结果的任务。

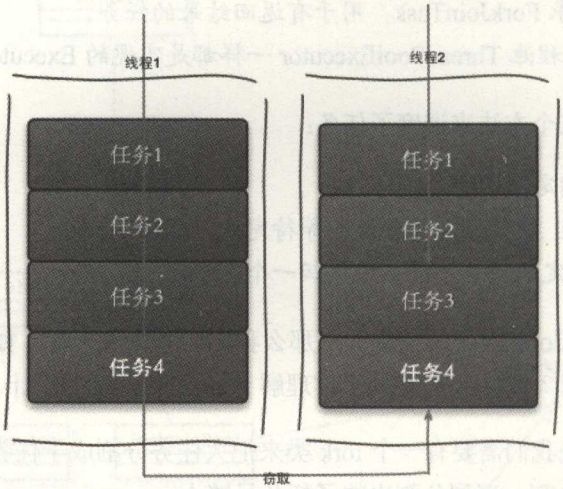
（2）ForkJoinPool：ForkJoinTask 需要通过 ForkJoinPool 来执行，任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列的头部。当一个工作线程的队列里暂时没有任务

时，它会随机从其他工作线程的队列的尾部获取一个任务。

7.4 Fork/Join 框架的实现原理

Fork/Join（分叉/结合）框架是一个比较特殊的线程池框架，专用于需要将一个任务不断分解成子任务（分叉），再不断进行汇总得到最终结果（结合）的计算过程。比起传统的线程池类 ThreadPoolExecutor，ForkJoinPool 实现了工作窃取算法，使得空闲线程能够主动分担从别的线程分解出来的子任务，从而让所有的线程都尽可能处于饱满的工作状态，并因此提高了执行效率。

工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。工作窃取的运行流程图如下：



那么为什么需要使用工作窃取算法呢？假如我们需要做一个比较大的任务，我们可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，于是把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应，比如 A 线程负责处理 A 队列里的任务。但是有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时，它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

工作窃取算法的优点是充分利用线程进行并行计算，并减少了线程间的竞争，其缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且消耗了更多的系统资源，比如创建多个线程和多个双端队列。

ForkJoinPool 由 ForkJoinTask 数组和 ForkJoinWorkerThread 数组组成，ForkJoinTask 数组负责存放程序提交给 ForkJoinPool 的任务，而 ForkJoinWorkerThread 数组负责执行这些任务。

ForkJoinTask 的 fork 方法实现原理。当我们调用 ForkJoinTask 的 fork 方法时，程序会调用 push 方法异步地执行这个任务，然后立即返回结果。代码如下：

```
public final ForkJoinTask<V> fork() {
        Thread t;
        if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread)
            ((ForkJoinWorkerThread)t). workQueue.push(this );
        else
            ForkJoinPool. common.externalPush(this);
        return this ;
}
```

push 方法把当前任务存放在 ForkJoinPool 数组 WorkQueue 里。

ForkJoinPool 里面的一些变量如下：

```
WorkQueue[] workQueues;                    // 工作队列
```

而 WorkQueue 又做了上面我们说的队列算法，一些变量如下：

```
@sun.misc.Contended
static final class WorkQueue {
        ForkJoinTask<?>[] array;
        final ForkJoinPool pool ;
        final ForkJoinWorkerThread owner ;
        volatile Thread parker ;
        volatile ForkJoinTask<?> currentJoin ;  // 正在被分隔的任务
        .....
}
```

ForkJoinTask 的 join 方法实现原理。Join 方法的主要作用是阻塞当前线程并等待获取结果。让我们一起看看 ForkJoinTask 的 join 方法的实现，代码如下：

```
public final V join() {
        int s;
        if ((s = doJoin() & DONE_MASK) != NORMAL)
            reportException(s);
        return getRawResult();
}
```

```
public final V getRawResult() {
        return result ;
}
```

首先，它调用了 doJoin()方法，通过 doJoin()方法得到当前任务的状态来判断返回什么结果，任务状态有 4 种：已完成（NORMAL），被取消（CANCELLED），信号（SIGNAL）和出现异常（EXCEPTIONAL）。

- 如果任务状态是已完成，则直接返回任务结果。
- 如果任务状态是被取消，则直接抛出 CancellationException。
- 如果任务状态是抛出异常，则直接抛出对应的异常。

让我们再来分析下 doJoin()方法的实现代码：

```
private int doJoin() {
        int s; Thread t; ForkJoinWorkerThread wt; ForkJoinPool.WorkQueue w;
        return (s = status ) < 0 ? s :
            ((t = Thread. currentThread()) instanceof ForkJoinWorkerThread) ?
            (w = (wt = (ForkJoinWorkerThread)t). workQueue).
            tryUnpush( this) && (s = doExec()) < 0 ? s :
            wt. pool.awaitJoin(w, this) :
            externalAwaitDone();
}
```

在 doJoin()方法里，首先通过查看任务的状态，看任务是否已经执行完了，如果执行完了，则直接返回任务状态，如果没有执行完，则从任务数组里取出任务并执行。如果任务顺利执行完成了，则设置任务状态为 NORMAL，如果出现异常，则记录异常，并将任务状态设置为 EXCEPTIONAL。

如果再往下面看，你会发现里面用了同步代码块锁的机制和线程 interrupt 阻塞机制，代码如下：

```
.....
boolean interrupted = false;
do {
   if (U .compareAndSwapInt(this, STATUS, s, s | SIGNAL)) {
                        synchronized (this ) {
                            if (status >= 0) {
                                try {
                                    wait();
                                } catch (InterruptedException ie) {
                                    interrupted = true;
                                }
```

```
                        }
                        else
                            notifyAll();
                    }
    }
} while ((s = status ) >= 0);
if (interrupted)
    Thread. currentThread().interrupt();
......
```

7.5 异常处理机制和办法

ForkJoinTask 在执行的时候可能会抛出异常，但是我们没办法在主线程里直接捕获异常，所以 ForkJoinTask 提供了 isCompletedAbnormally()方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过 ForkJoinTask 的 getException 方法获取异常。代码如下：

```
if(subTask.isCompletedAbnormally())
{
    System.out.println(subTask.getException());
}
```

getException 方法返回 Throwable 对象，如果任务被取消了，则返回 CancellationException。如果任务没有完成或者没有抛出异常，则返回 null。

7.6 Fork/Join 模式优缺点及其实际应用场景

通过使用 Fork/Join 模式，软件开发人员能够方便地利用多核平台的计算能力，轻松地实现并发程序的任务拆分。尽管还没有做到对软件开发人员完全透明，Fork/Join 模式已经极大地简化了编写并发程序的琐碎工作。对于符合 Fork/Join 模式的应用，软件开发人员不再需要处理各种并行相关事务，例如同步、通信等，以难以调试而闻名的死锁和 data race 等错误也就不会出现，提升了思考问题的层次。并行分发策略，仅仅关注如何划分任务和组合中间结果，将剩下的事情丢给 Fork/Join 框架完成即可。

唯一需要注意的是：如果拆分的对象过多时，小心一下子把内存撑满。等待线程的 CPU 资源释放了，但是线程对象等待时不会被垃圾机制回收。

我们来看一下 Fork/Join 模式的实际应用场景。对于树形结构类型的数据的处理和遍历非常

适合。比如：我们要对一个静态资源服务器的图片文件目录树进行遍历和分析的时候，我们需要递归地统计每个目录下的文件数量，最后汇总，非常适合用分叉/结合框架来处理。实例代码如下：

```
public class ForkJoinTaskDemo {
        public static void main(String[] args) throws InterruptedException,
ExecutionException {
            Integer count
= new ForkJoinPool().invoke(new CountingTask(Paths.get("D:/fish")));
            System. out.println("D:盘fish下面总文件数量: " +count);
            System. out.println("Thread Main End!" );
        }
}
// 处理单个目录的任务
class CountingTask extends RecursiveTask<Integer> {
        private Path dir ;
        public CountingTask(Path dir) {
            this.dir = dir;
        }
        @Override
        protected Integer compute() {
            int count = 0;
            List<CountingTask> subTasks = new ArrayList<CountingTask>();
            // 读取目录 dir 的子路径。
            try {
                DirectoryStream<Path> ds = Files.newDirectoryStream(dir);
                for (Path subPath : ds) {
                    if (Files.isDirectory(subPath,
LinkOption.NOFOLLOW_LINKS )) {
                        // 对每个子目录都新建一个子任务。
                        subTasks.add( new CountingTask(subPath));
                    } else {
                        // 遇到文件，则计数器增加 1。
                        count++;
                    }
                }
                if (!subTasks.isEmpty()) {
                    // 在当前的 ForkJoinPool 上调度所有的子任务。
                    for (CountingTask subTask : invokeAll(subTasks)) {
                        count += subTask.join();
```

```
                    }
                }
            } catch (IOException ex) {
                return 0;
            }
            return count;
        }
    }
```

运算结果如下，运算的速度还是非常快的，但是一旦文件多了，也是非常耗资源的，电脑就会出现卡顿的情况。

```
D:盘 fish 下面总文件数量：7647
Thread Main End!
```

第3部分

实际的使用、监控与拓展

第 8 章

线程、线程池在实际互联网项目开发中的应用

不登高山，不知山之高。不临深渊，不知地之厚也。

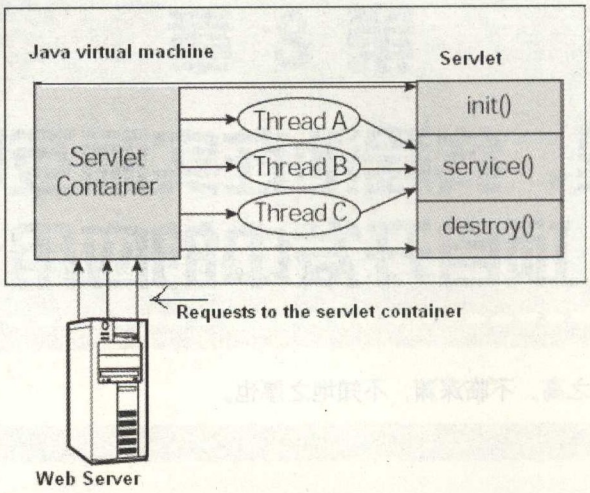
认识实际开发过程中的线程和线程池的相关知识与实战。

8.1 Servlet 线程的设计

一个客户端的浏览器请求，就是一个新的线程。

Server 端的服务器引擎都会建立一个 socket 连接监听浏览器发回来的用户请求，然后就把每一个请求当成一个新的线程来处理。

Servlet 生命周期分为三个阶段，如下图所示。



阶段 1：初始化阶段调用 init()方法。Servlet 在下列时刻进行初始化阶段。

- Servlet 容器启动时自动装载某些 Servlet，实现它只需要在 web.XML 文件中的<Servlet></Servlet>之间添加一行：<loadon-startup>1</loadon-startup>。
- 在 Servlet 容器启动后，客户首次向 Servlet 发送请求。
- Servlet 类文件被更新后，重新装载 Servlet，Servlet 被装载后，Servlet 容器创建一个 Servlet 实例并且调用 Servlet 的 init()方法进行初始化。在 Servlet 的整个生命周期内，init()方法只被调用一次。

阶段 2：响应客户请求阶段，调用 service()方法。

阶段 3：终止阶段。当服务器关闭的时候，调用 destroy()方法。

可见 Servlet 在整个 Tomcat 或者就 Jetty 的服务器中是单例的。所以，在共享变量的时候是线程不安全的，大家使用的时候要小心一点。但是，为什么相对于每个请求来说又是线程安全的呢？

先看如下 Servlet 源码：

```
public abstract interface Servlet
{
  public abstract void init(ServletConfig
paramServletConfig)throws ServletException;
  public abstract ServletConfig getServletConfig();
  public abstract void service(ServletRequest                paramServletRequest,
ServletResponse paramServletResponse)throws ServletException, IOException;
  public abstract String getServletInfo();
  public abstract void destroy();
}
```

我们看到只有 service 方法可以处理用户请求，每一次用户的请求都会创建 ServletRequest 的

一个新的对象，所以 ServletRequest 是线程安全的，对于每一个请求由一个工作线程来执行，所以 ServletResquest 只能在一个线程中被访问，而且它只在 service()方法内是有效的。

最后，既然每个请求都会发起一个线程，那么就会出现我们之前说的多线程并发的问题，而我们工作中呢，又是如何解决的呢？其实不论 Tomcat、Jetty，还是 Nginx，都有相应的解决方案，我们接下来详细看看。

8.2 线程池如何合理设计和配置

既然有了线程池，那怎么样才算配置设置合理呢？总结起来就是一句话，最大限度地发挥单台物理 server 机器的最大并发量（即：线程数），但又不至于服务器宕机，停止响应，而引起连锁反应形成系统崩溃。那么就必须知道两个参数了：本台 server 的最大可响应的并发量是多少？极限并发量是多少？接下来，我们将讲述两种测出最大并发线程的方法。

（1）第 1 章中我们讲到，如何计算一个计算机（物理机）的最大并发量，这个最大并发量其实就是这个 server 的最小并发量。而我们在第 1 章也说了 CPU 的线程工作机制及其和内存的关系，知道根据请求的大小和 CPU 的运行机制和执行时间，就可以算出一个大概的最大并发量。根据的参数有：内存大小和响应大小对应、宽带和请求大小对应、单台 IO 的运行时间、CPU 切换时间和真正的程序执行时间对应、有没有数据库处理上的等待。

（2）通过日志监控，看看用户都访问了哪些请求页面，发送了哪些参数，然后找一个和服务器一样配置的机器做一下压力测试就可以得出服务器的最大并发和抗压能力了。比如：携程的搜索服务器线上一共是四台做的负载均衡，而通过压力测试我们发现每个 server 的最大并发量是 300 个线程。

知道了这些信息之后，就可以通过我们前面讲的线程池的设置参数 int corePoolSize、int maximumPoolSize，设置一个合理的大小就可以了。

8.3 Tomcat 中线程池如何合理设置

（1）先来看 Tomcat 里面如何配置线程池。

首先，打开/conf/server.xml，增加如下代码：

```
<Executor name="tomcatThreadPool" namePrefix="catalina-exec-"
        maxThreads="200" minSpareThreads="25" maxIdleTime="60000" />
```

最大线程 200，最小空闲线程数 25，线程最大空闲时间 60s。

然后，修改<Connector ...>节点，增加 executor 属性，代码如下：

```
<Connector executor="tomcatThreadPool"
               port="80" protocol="HTTP/1.1"
               connectionTimeout="60000"
               keepAliveTimeout="15000"
               maxKeepAliveRequests="1"
               redirectPort="8443"
               ...../>
```

tomcatThreadPool 为上面线程池的名字。

可以多个 connector 共用 1 个线程池。

（2）接下来我们看一下 Executor 的参数有哪些，如下表所示：

参数	说明
className	线程池的实现类，如果自定义要实现 org.apache.catalina.Executor 接口。不填的情况下，默认 org.apache.catalina.core.StandardThreadExecutor
name	线程池的名字，当然了 server.xml 可以定义多个线程池
threadPriority	线程的优先级，默认 5
daemon	是否是守护线程，默认 true
namePrefix	线程的名字前缀，线程的名字有 namePrefix+threadNumber 组成。我们上面设置的是“catalina-exec-”
maxThreads	线程池最大的线程数量，默认 200。类似线程池里面的 corePoolSize
minSpareThreads	线程池的永远活动的线程的数量，默认 25。类似线程池里面的 maximumPoolSize
maxIdleTime	minSpareThreads 到 maxThreads 之前的线程最长存活时间，默认 60000ms
maxQueueSize	线程队列的最大值。我们前面不是说过 queue 吗，超过这个队列再进来的线程就会直接抛弃处理，默认：Integer.MAX_VALUE

由于我们本书主要讲解线程池，Connector 的参数我们就不在这里一一说明了（详细内容请参看附录 4）。

（3）Tomcat 的线程池 Executor 实现原理。

我们先看一下 Tomcat 容器里面 org.apache.catalina.Executor 的源码：

```
public abstract interface Executor extends java.util.concurrent.Executor,
Lifecycle
{
  public abstract String getName();
}
```

所以只要实现 Excecutor 接口就可以自定义线程池了。

我们再看一下默认 org.apache.catalina.core.StandardThreadExecutor 的源码：

```
public class StandardThreadExecutor implements Executor {
        protected int threadPriority = 5;//默认值
        protected boolean daemon = true;//默认值
        protected String namePrefix = "tomcat-exec-";//默认值
        protected int maxThreads = 200;//默认值
        protected int minSpareThreads = 25;//默认值
        protected int maxIdleTime = 60000;//默认值
        protected ThreadPoolExecutor executor = null;
        protected String name ;//线程池名称
        private LifecycleSupport lifecycle = new LifecycleSupport( this);
        //利用 Lifecycle 的循环机制开始方法
        public void start() throws LifecycleException {
              this.lifecycle .fireLifecycleEvent("before_start", null);
              //我们前面讲的 queue 机制，往下看的话是用的 LinkedBlockingQueue<Runnable>
机制。
             TaskQueue taskqueue = new TaskQueue();
              //这里用了我们前面讲的自定义线程工厂方法。
             TaskThreadFactory tf = new TaskThreadFactory(this .namePrefix );
              this.lifecycle .fireLifecycleEvent("start", null);
              //在这里用到了我们前面讲的自定义一个线程池，这里就可以很好的解释了我们前面的一
些参数的意思和用意了。
              this.executor = new ThreadPoolExecutor(getMinSpareThreads(),
getMaxThreads(), this .maxIdleTime , TimeUnit.MILLISECONDS, taskqueue, tf);
             taskqueue.setParent( this.executor );
              this.lifecycle .fireLifecycleEvent("after_start", null);
        }
        public void stop() throws LifecycleException {
              this.lifecycle .fireLifecycleEvent("before_stop", null);
              this.lifecycle .fireLifecycleEvent("stop", null);
              if (this .executor != null)
                    this.executor .shutdown();
              this.executor = null;
              this.lifecycle .fireLifecycleEvent("after_stop", null);
        }
        public void execute(Runnable command) {
              if (this .executor != null)
                    try {
```

```
                    this.executor .execute(command);
                } catch (RejectedExecutionException rx) {
                    if (!((TaskQueue) this.executor .getQueue()).force(co
mmand)))
                        throw new RejectedExecutionException();
                }
            else
                throw new IllegalStateException("StandardThreadPool      not
started.");
        }
    //……省略一些

    public int getPoolSize() {
        return ((this .executor != null) ? this.executor .getPoolSize() : 0);
    }

    public int getQueueSize() {
        return ((this .executor != null)? this.executor .getQueue().size(): -1);
    }
    //前面我们讲的线程工厂的实现方法
    class TaskThreadFactory implements ThreadFactory {
        final ThreadGroup group ;
        final AtomicInteger threadNumber = new AtomicInteger(1);
        final String namePrefix ;
        TaskThreadFactory(String namePrefix) {
            SecurityManager s = System. getSecurityManager();
            this.group =    ((s    != null)    ?    s.getThreadGroup()    :
Thread.currentThread().getThreadGroup());
            this.namePrefix = namePrefix;
        }
        public Thread newThread(Runnable r) {
            Thread                t                = new Thread(this.group,
r, this.namePrefix + this.threadNumber .getAndIncrement());
            t.setDaemon(StandardThreadExecutor. this.daemon );
            t.setPriority(StandardThreadExecutor. this.getThreadPriority
());
            return t;
        }
    }
```

```
        //前面我们讲的 LinkedBlockingQueue 的实现方法
          class TaskQueue extends LinkedBlockingQueue<Runnable> {
                ThreadPoolExecutor parent = null ;
                 public TaskQueue() {}
                 public TaskQueue(int paramInt) {
                       super(paramInt);
                 }
                 public void setParent(ThreadPoolExecutor tp) {
                       this.parent = tp;
                 }
                 public boolean force(Runnable o) {
                       if (this .parent .isShutdown())
                             throw new RejectedExecutionException("Executor      not
running, can't force a command into the queue");
                       return super .offer(o);
                 }
                 public boolean offer(Runnable o) {
                       if (this .parent == null)
                             return super .offer(o);
                       if (this .parent .getPoolSize()
== this.parent .getMaximumPoolSize())
                             return super .offer(o);
                       if (this .parent .getActiveCount()
< this.parent .getPoolSize())
                             return super .offer(o);
                       if (this .parent .getPoolSize()
< this.parent .getMaximumPoolSize())
                             return false ;
                       return super .offer(o);
                 }
          }
    }
```

（4）局限性：Tomcat 的唯一局限性，就是最多只能发挥一台物理机器的最大性能，而且达不到这台物理机器的极限，因为我们要处理程序、处理 IO 等等。而如何分布式，如何集群呢，我们接下来以 Nginx 为例来讲解一下。

8.4 Nginx 线程池

Nginx 的最大作用就是做负载均衡，进行请求的分发，不要做 IO，不要做其他任何分占 CPU 和内存的事情，只做负载分发，Nginx 本身设计得也很好，基本上可以接近 CPU 原生的线程并发峰值，好的服务器配置好的话，应该 3 秒内并发个几千没什么问题。接下来我们说一下 Nginx 的常用的配置，以及如何设置并发？

其实 Nginx 里面没有给定线程池的相关概念和配置，但是如果为了以防万一的话，可以注意如下参数：

（1）ngx_http_limit_req_module 模块（0.7.21）可以通过定义的键值来限制请求处理的频率。特别地，它可以限制来自单个 IP 地址的请求处理频率。限制的方法是通过一种“漏桶”的方法——固定每秒处理的请求数，推迟过多的请求处理。

```
语法:limit_req zone=name [burst=number] [nodelay];
默认值: —
上下文: http, server, location
```

设置对应的共享内存限制域和允许被处理的最大请求数阈值。 如果请求的频率超过了限制域配置的值，请求处理会被延迟，所以，所有的请求都是以定义的频率被处理的。超过频率限制的请求会被延迟，直到被延迟的请求数超过了定义的阈值，这时，这个请求会被终止，并返回 503（Service Temporarily Unavailable）错误。这个阈值的默认值等于 0。比如下面这些指令：

```
limit_req_zone $binary_remote_addr zone=one:10m rate=3000r/s;
server {
    location /search/ {
        limit_req zone=one burst=100;
    }
.....
}
```

限制平均每秒不超过 3 000 一个请求，同时允许超过频率限制的请求数不多于 100 个。

如果不希望超过的请求被延迟，可以用 nodelay 参数：

```
limit_req zone=one burst=100 nodelay;
```

（2）ngx_http_limit_conn_module 模块可以按照定义的键限定每个键值的连接数。特别地，可以设定单一 IP 来源的连接数。并不是所有的连接都会被模块计数，只有那些正在被处理的请求（这些请求的头信息已被完全读入）所在的连接才会被计数。

```
语法:limit_conn zone number;
```

```
默认值:—
上下文:http, server, location
```

指定一块已经设定的共享内存空间，以及每个给定键值的最大连接数。当连接数超过最大连接数时，服务器将会返回 503（Service Temporarily Unavailable）错误。比如，下面配置不仅会限制单一 IP 来源的连接数，同时也会限制单一虚拟服务器的总连接数：

```
limit_conn_zone $binary_remote_addr zone=perip:10m;
limit_conn_zone $server_name zone=perserver:10m;
server {
    ...
    limit_conn perip 10;
    limit_conn perserver 200;
}
```

上面这个配置表示，同一 IP 同一时间只允许有 10 个连接。单一虚拟服务器的总连接数 200。

接下来我们看一下这种线程池的局限性。线程池配置好以后，现在的并发请求局限就是 Nginx 服务器了，那接下来我们怎么做到淘宝、京东、携程那种大并发的数量级别呢？我们看看下面两种做法：

第一种：高富帅的玩法，就是最常见的投入成本，见效快，运维简单的 A10、F5 的硬件负载均衡器，直接进行请求转发。这时候基本上不需要 Nginx，直接到后台 Server 了。当然了加 Nginx 更好，可以再扩大 N 倍。

第二种：屌丝级的玩法，就是通过域名解析，建立很多子域名，不同的业务模块由域名分发到不同的 Nginx 的服务器上面去，每个 Nginx 再都配置上内容分发网络（CDN），实行静态分离，尽量给 Nginx 服务器减少负担。这种配置基本上可以解决很多中小型公司的问题。

8.5 数据库连接池

其实我们说了这么多的线程和线程池，数据库连接池也是一样的。

（1）什么是连接

连接，是我们的编程语言与数据库交互的一种方式。我们经常会听到这么一句话“数据库连接很昂贵”。有人接受这种说法，却不知道它的真正含义。因此，下面我将解释它究竟是什么（如果你已经知道了，你可以跳到它的工作原理部分）。

（2）创建连接的代码片段：

```
String connUrl = "jdbc:mysql://your.database.domain/yourDBname";
```

```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection (connUrl);
```

当我们创建了一个 Connection 对象，它在内部都执行了什么：

- DriverManager 检查并注册驱动程序。
- com.mysql.jdbc.Driver 就是我们注册了的驱动程序，它会在驱动程序类中调用 connect(url…)方法。
- com.mysql.jdbc.Driver 的 connect 方法根据我们请求的 connUrl，创建一个 Socket 连接，连接到 IP 为 your.database.domain，默认端口 3306 的数据库。
- 创建的 Socket 连接将被用来查询我们指定的数据库，并最终让程序返回得到一个结果。

（3）为什么昂贵

现在让我们谈谈为什么说它“昂贵”。如果创建 Socket 连接花费的时间比实际的执行查询的操作所花费的时间还要更长。这就是我们所说的“数据库连接很昂贵”，因为连接资源数是 1，它需要每次创建一个 Socket 连接来访问 DB。因此，我们将使用连接池。连接池初始化时创建一定数量的连接，然后从连接池中重用连接，而不是每次创建一个新的。

（4）为什么在连接数据库时要使用连接池

数据库连接是一种关键的、有限的昂贵资源，这一点在多用户的网页应用程序中体现得尤为突出。一个数据库连接对象均对应一个物理数据库连接，每次操作都打开一个物理连接，使用完都关闭连接，这样造成系统的性能低下。数据库连接池的解决方案是，在应用程序启动时建立足够的数据库连接，并把这些连接组成一个连接池（简单说就是在一个“池”里放了好多半成品的数据库联接对象），由应用程序动态地对池中的连接进行申请、使用和释放。对于多于连接池中连接数的并发请求，应该在请求队列中排队等待。并且应用程序可以根据池中连接的使用率，动态增加或减少池中的连接数。连接池技术尽可能多地重用了消耗内存的资源，大大节省了内存，提高了服务器的服务效率，能够支持更多的客户服务。通过使用连接池，将大大提高程序运行效率，同时，我们可以通过其自身的管理机制来监视数据库连接的数量、使用情况等。

（5）数据库连接池的基本原理

数据库连接池的基本思想就是为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去。我们可以通过设定连接池最大连接数，来防止系统无尽地与数据库连接。更为重要的是，我们可以通过连接池的管理机制监视数据库的连接数量、使用情况，为系统开发、测试及性能调整提供依据。

（6）连接池的工作原理主要由三部分组成，分别为连接池的建立、连接池中连接的使用管理、连接池的关闭。

第一，连接池的建立。一般在系统初始化时，连接池会根据系统配置建立，并在池中创建了几个连接对象，以便使用时能从连接池中获取。连接池中的连接不能随意创建和关闭，这样避免

了连接随意建立和关闭造成的系统开销。Java 中提供了很多容器类可以方便地构建连接池，例如 Vector、Stack 等。

第二，连接池的管理。连接池管理策略是连接池机制的核心，连接池内连接的分配和释放对系统的性能有很大的影响。其管理策略是：

当客户请求数据库连接时，首先查看连接池中是否有空闲连接，如果存在空闲连接，则将连接分配给客户使用；如果没有空闲连接，则查看当前所开的连接数是否已经达到最大连接数，如果没达到就重新创建一个连接给请求的客户；如果达到就按设定的最大等待时间进行等待，如果超出最大等待时间，则抛出异常给客户。

当客户释放数据库连接时，先判断该连接的引用次数是否超过了规定值，如果超过就从连接池中删除该连接，否则保留为其他客户服务。

该策略保证了数据库连接的有效复用，避免频繁地建立、释放连接所带来的系统资源开销。

第三，连接池的关闭。当应用程序退出时，关闭连接池中所有的连接，释放连接池相关的资源，该过程正好与创建相反。

（7）很多国内 Java 开发人员都推荐使用 Alibaba 集团开发人员开发的开源项目 Druid，官方说是 Java 语言中最好的数据库连接池。Druid 能够提供强大的监控和扩展功能。

以下是一个配置_DruidDataSource 参考的连接池配置：

```
   <bean id= "dataSource" class ="com.alibaba.druid.pool.DruidDataSource" init
-method="init" destroy-method= "close">
          <!-- 基本属性 url、user、password -->
          <property name= "url" value ="${jdbc_url}" />
          <property name= "username" value ="${jdbc_user}" />
          <property name= "password" value ="${jdbc_password}" />
          <!-- 配置初始化大小、最小、最大 -->
          <property name= "initialSize" value ="1" />
          <property name= "minIdle" value ="1" />
          <property name= "maxActive" value ="20" />
          <!-- 配置获取连接等待超时的时间 -->
          <property name= "maxWait" value ="60000" />
          <!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是 ms -->
          <property name= "timeBetweenEvictionRunsMillis" value="60000" />
          <!-- 配置一个连接在池中最小生存的时间，单位是 ms -->
          <property name= "minEvictableIdleTimeMillis" value="300000" />
          <!-- 打开 PSCache，并且指定每个连接上 PSCache 的大小 -->
          <property name= "poolPreparedStatements" value="true" />
          <property name= "maxPoolPreparedStatementPerConnectionSize" value="2
0" />
          <!-- 配置监控统计拦截的 filters -->
```

```
        <property name= "filters" value ="stat" />
</bean>
```

通常来说，只需要修改 initialSize、minIdle、maxActive。

如果用 Oracle，则把 poolPreparedStatements 配置为 true，mysql 可以配置为 false。分库分表较多的数据库，建议配置为 false。

（8）我们如果查看一下 com.alibaba.druid.pool.DruidDataSource 的源码的话，牛人们也不外乎使用了 CountDownLatch 做线程阀，AtomicLong 的高并发原子变量，ReentrantLock 显示块状线程锁，volatile 关键字做高并发的可读变量，DruidConnectionHolder[] connections 储存连接队列，ThreadLocal 做线程安全的副本。

8.6 如何在分布式系统中实现高并发

在工作中，如何分布系统，如何实现高并发系统的前提条件就是：

- 了解你的物理机器
- 了解你的业务
- 了解你的程序

说到最后就是一句拆分，再拆分，接下来我们讲一些拆分的原则。

首先分析任务特性，可以从以下几个角度来进行分析：

（1）任务的性质：CPU（计算密集型）任务，IO（网络 IO，DB，磁盘 IO 等）密集型任务，内存消耗性。

（2）任务的并发数量：超级大并发，大并发，中等量，很少人访问等。

（3）任务的执行时间：长，中，短。

（4）任务的优先级：高，中，低。

（5）任务的依赖性：是否依赖其他系统资源，如数据库连接，业务流程之间是否互相依赖等。

（6）按照资源性质：静态资源，动态资源。

（7）业务之间的耦合性：耦合度高，可解耦，没有耦合。

当我们对我们的业务和任务有了全面的认识之后，就可以合理地配置线程池，从而实现高可用、高并发，原则如下：

（1）任务性质不同的任务可以用不同规模的线程池分开处理。CPU 密集型任务配置尽可能小的线程，如配置 Ncpu+1 个线程的线程池。IO 密集型任务则由于线程并不是一直在执行任务，则配置尽可能多的线程，如 2*Ncpu。混合型的任务，如果可以拆分，则将其拆分成一个 CPU 密集型任务和一个 IO 密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的

吞吐率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过 Runtime.getRuntime().availableProcessors()方法获得当前设备的 CPU 个数。

（2）优先级不同的任务可以使用优先级队列 PriorityBlockingQueue 来处理。它可以让优先级高的任务先得到执行，需要注意的是，如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远得不到执行。

（3）执行时间不同的任务可以交给不同规模的线程池来处理，或者也可以使用优先级队列，让执行时间短的任务先执行。

（4）依赖数据库连接池的任务，因为线程提交 SQL 后需要等待数据库返回结果，如果等待的时间越长，CPU 空闲时间就越长，那么线程数应该设置越大，这样才能更好地利用 CPU。

（5）建议使用有界队列，有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点，比如几千。有一次我们组使用的后台任务线程池的队列和线程池全满了，不断地抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行 SQL 变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞住，任务积压在线程池里。如果当时我们设置成无界队列，线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。当然我们的系统所有的任务是用的单独的服务器部署的，而我们使用不同规模的线程池跑不同类型的任务，但是出现这样问题时也会影响到其他任务。

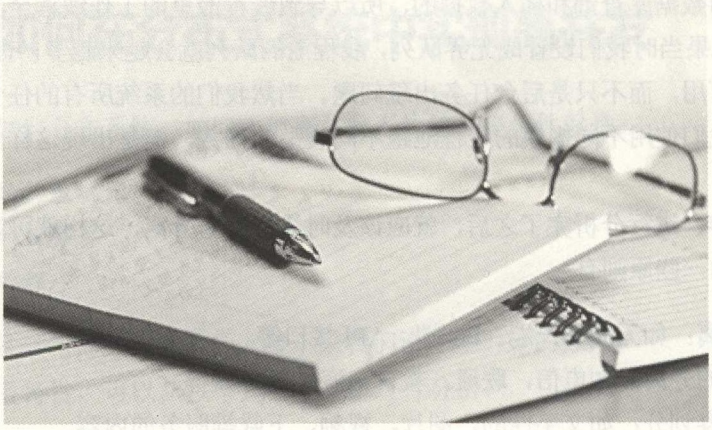
对服务的内容，进行分析完了之后，就应该及时进行部署分离，这样就可以实现高性能的分布式，部署分离的一些原则有：

（1）渠道分离：如无线客户端，PC 端，API 接口等。
（2）运营商的分离：如电信，联通，教育，海外服务器等。
（3）服务内容划分：如文字性的，图片，视频，下载等服务的内容。
（4）按照访问密集型性划分：如高并发的和低并发的可以分开部署在不同的环境下。
（5）按照不同业务线进行划分：如前台项目，BBS，后台管理项目等。

最后，在你全面了解你的业务，你的 Server，你的程序的基础上通过分发配置，还有线程池的过载机制配置，这个时候你就可以做到泰山崩于前而色不变，麋鹿兴于左而目不瞬了。

第 9 章
线程的监控及其日常工作中如何分析

鸟欲高飞先振翅，人求上进先读书。



看不到不等于不存在！让我们来看看工作中是如何找问题解决问题的。

9.1 Java 线程池的监控

如果想实现线程池的监控，需要自定义线程池继承 ThreadPoolExecutor 类，并且实现 beforeExecute、afterExecute 和 terminated 方法，我们可以在任务执行前、执行后和线程池关闭前干一些事情。比如，监控任务的平均执行时间，最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。请看下面代码：

```
//每执行一个工作任务线程之前都会执行此实现的方法
protected void beforeExecute(Thread t, Runnable r) {
//t - 放在线程池里面要执行的线程。
//r - 将要执行这个线程的线程池里面的工作线程。
```

```
    }

    //每执行一个工作任务线程之后都会执行的方法
    protected void afterExecute(Runnable r, Throwable t) {
    //r - 已经运行结束的工作线程。
    //t - 运行异常。
    }

    //线程池关闭之前可以干一些事情。
    protected void terminated() { };
```

线程池里有一些属性在监控线程池的时候可以使用：

- taskCount：线程池需要执行的任务数量。
- completedTaskCount：线程池在运行过程中已完成的任务数量。小于或等于 taskCount。
- largestPoolSize：线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过。如等于线程池的最大大小，则表示线程池曾经满了。
- getPoolSize：线程池的线程数量。如果线程池不销毁的话，池里的线程不会自动销毁，所以这个大小只增不减。
- getActiveCount：获取活动的线程数。

大家想一想如果你来写的话如何去写，提供实例 demo 如下，慢慢体会一下：

```
public class MonitorThreadPoolExecutorDemo {
        public static void main(String[]       args) throws InterruptedException,
ExecutionException {
            Thread. sleep(500L);// 方便测试
            ExecutorService executor = new MonitorThreadPoolExecutor(5, 5, 0L,
TimeUnit.MILLISECONDS , new LinkedBlockingQueue() );
             for (int i = 0; i < 3; i++) {
                  Runnable runnable = new Runnable() {
                        public void run() {
                              try {
                                    Thread. sleep(100L);
                              } catch (InterruptedException e) {
                                    e.printStackTrace();
                              }
                        }
                  };
                  executor.execute(runnable);
```

```
            }
            executor.shutdown();
            System. out.println("Thread Main End!" );
        }
    }
    class MonitorThreadPoolExecutor extends ThreadPoolExecutor {
        public MonitorThreadPoolExecutor(int arg0, int arg1, long arg2,  TimeUnit
arg3, BlockingQueue<Runnable> arg4) {
            super(arg0, arg1, arg2, arg3, arg4);
        }
        protected void beforeExecute(Thread paramThread, Runnable paramRunnable)
{
            System. out.println("work_task before:" + paramThread.getName());
        }
        protected void afterExecute(Runnable r, Throwable t) {
            super.afterExecute(r, t);
            System. out.println("work_task after worker thread is :" + r);
        }
        protected void terminated() {
            System. out.println("terminated
getCorePoolSize:" + this.getCorePoolSize() + "; getPoolSize:" + this.getPoolSize()
+ "; getTaskCount:" + this .getTaskCount() + "; getCompletedTaskCount:"
                    + this.getCompletedTaskCount()          + "          ;
getLargestPoolSize:" + this.getLargestPoolSize()                + "          ;
getActiveCount:" + this.getActiveCount());
            System. out.println("ThreadPoolExecutor terminated:" );
        }
    }
```

运行结果如下：

```
    work_task before:pool-1-thread-1
    work_task before:pool-1-thread-3
    work_task before:pool-1-thread-2
    Thread Main End!
    work_task                after                worker                thread
is :demo.thread.MonitorThreadPoolExecutorDemo$1@13bbe97
    work_task                after                worker                thread
is :demo.thread.MonitorThreadPoolExecutorDemo$1@70125b
    work_task                after                worker                thread
```

```
is :demo.thread.MonitorThreadPoolExecutorDemo$1@ceba90
    terminated    getCorePoolSize:5    ;    getPoolSize:0    ;    getTaskCount:3    ;
getCompletedTaskCount:3; getLargestPoolSize:3; getActiveCount:0
    ThreadPoolExecutor terminated:
```

9.2 ForkJoin 如何监控

其实 ForkJoin 的监控不像 ThreadPoolExecutor 那样提供了 before 和 after 的方法，只是 ForkJoinPool 提供了一些可以查阅其状态信息的方法，如下：

- getPoolSize(): 此方法返回 int 值，它是 ForkJoinPool 内部线程池的 worker 线程们的数量。
- getParallelism(): 此方法返回池的并行的级别。
- getActiveThreadCount(): 此方法返回当前执行任务的线程的数量。
- getRunningThreadCount():此方法返回没有被任何同步机制阻塞的正在工作的线程。
- getQueuedSubmissionCount(): 此方法返回已经提交给池还没有开始他们的执行的任务数。
- getQueuedTaskCount(): 此方法返回已经提交给池已经开始它们的执行的任务数。
- hasQueuedSubmissions(): 此方法返回 Boolean 值，表明这个池是否有 queued 任务还没有开始它们的执行。
- getStealCount(): 此方法返回 long 值，worker 线程已经从另一个线程偷取到的时间数。
- isTerminated(): 此方法返回 Boolean 值，表明 fork/join 池是否已经完成执行。

以上方法更有助于我们开发过程中更加了解你的 forkjoin 的设计是否合理。

其实，实现这样的监控比较简单了，大家写一个试试。简单来看一下 Demo，我们将第 7 章的例子修改如下，增加一个 showLog 方法用来显示监控的信息：

```
  public class ForkJoinTaskDemo {
        public static void main(String[]       args) throws InterruptedException,
ExecutionException {
              ForkJoinPool forkJoinPool = new ForkJoinPool();
              CountTask task = new CountTask(1, 10);
              Future<Integer> result = forkJoinPool.submit(task);
              System. out.printf("Fork/Join     Pool:     Terminated     :%s\n" ,
forkJoinPool.isTerminated());
              System. out.println("最终的结果: " + result.get());
               showLog(forkJoinPool);
              System. out.println("Thread Main End!" );
        }
```

```
        // 接收 ForkJoinPool 对象作为参数和写关于线程和任务的执行的状态的信息。
        private static void showLog(ForkJoinPool join ) {
            System. out.printf("**********************\n" );
            System. out.printf("Fork/Join                                    Pool:
Parallelism:%d\n" , join.getParallelism());
            System. out.printf("Fork/Join                Pool:                Pool
Size:%d\n" , join.getPoolSize());
            System. out.printf("Fork/Join        Pool:        Active        Thread
Count:%d\n", join .getActiveThreadCount());
            System. out.printf("Fork/Join        Pool:        Running        Thread
Count:%d\n", join .getRunningThreadCount());
            System. out.printf("Fork/Join                Pool:                Queued
Submission:%d\n" , join.getQueuedSubmissionCount());
            System. out.printf("Fork/Join                Pool:                Queued
Tasks:%d\n" , join.getQueuedTaskCount());
            System. out.printf("Fork/Join                Pool:                Queued
Submissions:%s\n" , join.hasQueuedSubmissions());
            System. out.printf("Fork/Join                Pool:                Steal
Count:%d\n" , join.getStealCount());
            System. out.printf("Fork/Join                                    Pool:
Terminated :%s\n" , join.isTerminated());
            System. out.printf("**********************\n" );
        }
    }
```

运行结果如下：

```
Fork/Join Pool: Terminated :false
最终的结果：55
**********************
Fork/Join Pool: Parallelism:2
Fork/Join Pool: Pool Size:2
Fork/Join Pool: Active Thread Count:0
Fork/Join Pool: Running Thread Count:0
Fork/Join Pool: Queued Submission:0
Fork/Join Pool: Queued Tasks:0
Fork/Join Pool: Queued Submissions:false
Fork/Join Pool: Steal Count:1
Fork/Join Pool: Terminated :false
**********************
```

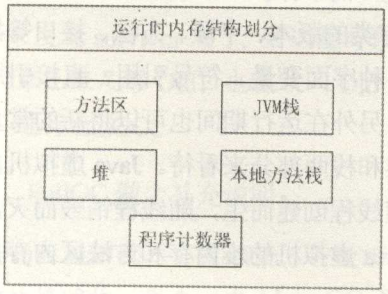
```
Thread Main End!
```

9.3 Java 内存结构

后面我们会讲到线程 Dump 的分析，那就不得不对 Java 里面的内存结构有一定的了解。其实对于 Java 程序员来说，在虚拟机的自动内存管理机制（这种机制又称为垃圾回收机制、garbage collector、GC）的帮助下，不再需要显式地为每一个 new 操作去分配内存，回收内存，而且通常情况不容易出现内存泄漏和内存溢出问题，看起来由虚拟机管理内存一切都很美好。不过，在高并发的开发模式下，不知不觉就会浪费很多内存，导致的后果就是 GC 回收执行时间太频繁，内存泄露，进而导致内存溢出，也正是因为 Java 程序员把内存控制的权力交给了 Java 虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那排查错误将会成为一项异常艰难的工作。

1. JVM 的内存结构

JVM 的内存结构其实大体上分成了这么几个部分：程序计数器、JVM 栈、本地方法栈、共享堆、方法区。如下图所示。



（1）程序计数器：是一块较小的内存空间，其作用可以看作是当前线程所执行的字节码的行号指示器，字节码解析器工作时，通过改变程序计数器的值来选取下一条需要执行的字节码指令。Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间片来实现，在任何一个时刻，一个处理器只会执行一条线程指令，因此，为了确保线程切换之后能恢复到正确的执行位置，每条线程都需要一个独立的程序计数器，因此，程序计数器是线程私有的内存。程序计数器是 Java 虚拟机中唯一一个没有规定任何内存溢出 OutOfMemoryError 的内存区域。

（2）JVM 栈：Java 虚拟机栈，也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是 java 方法执行的内存模型：每个方法被执行时都会同时创建一个栈帧用于存放局部变量表、操作数栈、动态连接和方法出口等信息。每个方法被调用直至执行完成过程，就对应着一个栈帧在虚拟机中从入栈到出栈的过程。Java 虚拟机栈的局部变量表存放了编译器可知的 8 种 Java 基本类型数据、对象引用（注意不是对象实例本身）、方法返回地址 returnAddress。局部变量表所需内存空间在编译期间完成分配，当进入一个方法时，该方法需要在帧中分配多大的局部变量空间是

完全确定的，在方法运行期间不会改变局部变量表的大小。Java 虚拟机栈有两种异常状况：如果单线程请求的栈深度大于虚拟机所允许的最大深度时，抛出 StackOverflowError 异常；如果虚拟机栈可以动态扩展，当扩展时无法申请到足够内存时会抛出 OutOfMemoryError 异常。

（3）本地方法栈：本地方法栈与 Java 虚拟机栈作用非常类似，其区别是：java 虚拟机栈是为虚拟机执行 java 方法服务，而本地方法栈是为虚拟机调用的操作系统本地方法服务。Java 虚拟机规范没有对本地方法栈的实现和数据结构做强制规定，Sun HotSpot 虚拟机直接把 Java 虚拟机栈和本地方法栈合二为一。

（4）共享堆：堆是 java 虚拟机所管理的内存区域中最大一块，Java 堆是被所有线程所共享的一块内存区域，在 Java 虚拟机启动时创建，堆内存的唯一目的就是存放对象实例。几乎所有的对象实例都是在堆分配内存。Java 堆是垃圾收集器管理的主要区域，从垃圾回收的角度看，由于现在的垃圾收集器基本都采用的是分代收集算法，因此 Java 堆还可以初步细分为新生代和年老代。Java 虚拟机规范规定，堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。在实现上既可以是固定大小的，也可以是可动态扩展的。如果在堆中没有内存完成实例分配，并且堆大小也无法再扩展时，将会抛出 OutOfMemoryError 异常。

（5）方法区：方法区与堆一样，是被各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是方法区却有一个别名叫 Non-Heap（非堆）。

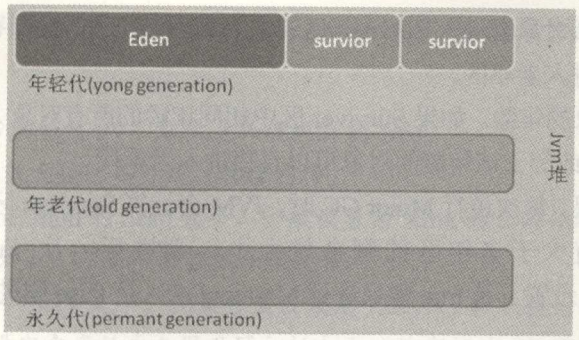
Sun HotSpot 虚拟机把方法区叫永久代（Permanent Generation），方法区中最重要的部分是运行时常量池。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面变量、符号引用、直接引用等，这些内容将在类加载后存放到方法区的运行时常量池中，另外在运行期间也可以将新的常量存放到常量池中。

其实，我更愿意将它分为堆和栈两部分来看待。Java 虚拟机内存结构中的程序计数器、虚拟机栈和本地方法栈这三个区域随线程创建而生，随线程销毁而灭，因此这三个区域的内存分配和回收是确定的生命周期的，而 Java 虚拟机的堆内存和方法区内存是通过垃圾回收机制三代关系来完成的，接下来我们来说一下。

2. JVM 的垃圾回收机制

针对垃圾回收算法将 JVM 中堆和非堆空间划分为三个代：年轻代（Young Generation）、年老代（Old Generation）和永久代（Permanent Generation），如下图所示。年轻代和老年代就是我们上面说的共享堆，年轻带主要是动态的存储，年轻带主要储存新产生的对象；年老代储存年龄大些的对象；永久带就是我们上面说的方法区，主要存储的是 Java 的类信息，包括解析得到的方法、属性、字段等，永久带基本不参与垃圾回收。所以我们说的垃圾回收主要是针对年轻代和年老代。

年轻代又分成 3 个部分，一个 eden 区和两个相同的 survior 区。刚开始创建的对象都是放置在 eden 区的。分成这样 3 个部分，主要是为了生命周期短的对象尽量留在年轻带。当 eden 区申请不到空间的时候，进行 minorGC，把存活的对象拷贝到 survior。



年老代主要存放生命周期比较长的对象，比如缓存对象。

3. JVM 内存垃圾回收过程

JVM 内存垃圾回收过程描述如下 ：

步骤 01　对象在 Eden 区完成内存分配。

步骤 02　当 Eden 区满了，再创建对象，会因为申请不到空间，触发 minorGC，进行 young（eden+1survivor）区的垃圾回收。

步骤 03　minorGC 时，Eden 不能被回收的对象被放入到空的 survivor（Eden 肯定会被清空），另一个 survivor 里不能被 GC 回收的对象也会被放入这个 survivor，始终保证一个 survivor 是空的

步骤 04　当做第 3 步的时候，如果发现 survivor 满了，则这些对象被 copy 到 old 区，或者 survivor 并没有满，但是有些对象已经足够 Old，也被放入 Old 区 XX:MaxTenuringThreshold。

步骤 05　当 Old 区被放满之后，进行 fullGC。

下面对 minorGC、MajorGC、FullGC 做个补充说明：

- MinorGC：年轻代所进行的垃圾回收，非常频繁，一般回收速度也比较快。
- MajorGC：老年代进行的垃圾回收，发生一次 MajorGC 至少伴随一次 MinorGC，一般比 MinorGC 速度慢十倍以上。
- FullGC：整个堆内存进行的垃圾回收，很多时候是 MajorGC。

MajorGC 和 FullGC 是最耗 CPU 的，会影响程序的响应，所以要根据相应原则合理设置年轻代和老年代的大小。所以监控的时候就会发现年轻代会有频繁的回收，而 FullGc 执行次数很少。一旦 FullGc 频繁发生的时候，基本上就快要内存溢出了。

4. JVM 内存垃圾回收过程中对象分配的原则

（1）对象优先分配在 Eden 区，如果 Eden 区没有足够的空间时，虚拟机执行一次 Minor GC。

（2）大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在 Eden 区和两个 Survivor 区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。

（3）长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过

了 1 次 Minor GC，那么对象会进入 Survivor 区；之后每经过一次 Minor GC，那么对象的年龄加 1，直到达到阀值对象进入老年区。

（4）动态判断对象的年龄。如果 Survivor 区中相同年龄的所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。

（5）空间分配担保。每次进行 Minor GC 时，JVM 会计算 Survivor 区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小，则进行一次 FullGC；如果小于检查 HandlePromotionFailure 设置，是 true 则只进行 Monitor GC，是 false 则进行 Full GC。

5. JVM 内存垃圾回收机制主要有三种

- 串行收集器：使用单线程处理所有垃圾回收工作，因为无须多线程交互，所以效率比较高。
- 并行收集器：对年轻代进行并行垃圾回收，因此可以减少垃圾回收时间。一般在多线程多处理器机器上使用。
- 并发收集器：可以保证大部分工作都并发进行（应用不停止），垃圾回收只暂停很少的时间，此收集器适合对响应时间要求比较高的中、大规模应用。

6. 常见的内存溢出的三种情况

（1）JVM Heap（堆）溢出：java.lang.OutOfMemoryError: Java heap space。

JVM 在启动的时候会自动设置 JVM Heap 的值，可以利用 JVM 提供的-Xmn -Xms -Xmx 等选项进行设置。Heap 的大小是 Young Generation 和 Tenured Generaion 之和。在 JVM 中如果 98%的时间是用于 GC，且可用的 Heap size 不足 2%的时候将抛出此异常信息。

解决方法：手动设置 JVM Heap（堆）的大小。

（2）PermGen space 溢出： java.lang.OutOfMemoryError: PermGen space 。

PermGen space 的全称是 Permanent Generation space，是指内存的永久保存区域。为什么会内存溢出，这是由于这块内存主要是被 JVM 存放 Class 和 Meta 信息的，Class 在被 Load 的时候被放入 PermGen space 区域，它和存放 Instance 的 Heap 区域不同，sun 的 GC 不会在主程序运行期对 PermGen space 进行清理，所以如果你的 APP 会载入很多 CLASS 的话，就很可能出现 PermGen space 溢出。一般发生在程序的启动阶段。

解决方法：通过-XX:PermSize 和-XX:MaxPermSize 设置永久代大小即可。

（3）栈溢出： java.lang.StackOverflowError : Thread Stack space。

栈溢出了，JVM 依然是采用栈式的虚拟机，这个和 C 和 Pascal 都是一样的。函数的调用过程都体现在堆栈和退栈上了。调用构造函数的 "层"太多了，以致于把栈区溢出了。 通常来讲，一般栈区远远小于堆区的，因为函数调用过程往往不会多于上千层，而即便每个函数调用需要 1KB 的空间（这个大约相当于在一个 C 函数内声明了 256 个 int 类型的变量），那么栈区也不过是需要 1MB 的空间。通常栈的大小是 1~2MB 的。通俗一点讲就是单线程的程序需要的内存太大了。通常递归也不要递归的层次过多，很容易溢出。

解决方法：修改程序，或者通过 -Xss 来设置每个线程的 Stack 大小即可。

（4） 所以 Server 容器启动的时候我们经常关心和设置 JVM 的几个参数如下（详细的 JVM 参数请参看附录 3）：

- -Xms：java Heap 初始大小， 默认是物理内存的 1/64。
- -Xmx：ava Heap 最大值，不可超过物理内存。
- -Xmn：young generation 的 heap 大小，一般设置为 Xmx 的三四分之一 。增大年轻代后，将会减小年老代大小，可以根据监控合理设置。
- -Xss：每个线程的 Stack 大小，而最佳值应该是 128KB，默认值好像是 512KB。
- -XX:PermSize：设定内存的永久保存区初始大小，缺省值为 64MB。
- -XX:MaxPermSize：设定内存的永久保存区最大大小，缺省值为 64MB。
- -XX:SurvivorRatio：Eden 区与 Survivor 区的大小比值，设置为 8，则两个 Survivor 区与一个 Eden 区的比值为 2:8，一个 Survivor 区占整个年轻代的 1/10。
- -XX:+UseParallelGC：F 年轻代使用并发收集，而年老代仍旧使用串行收集。
- -XX:+UseParNewGC：设置年轻代为并行收集，JDK5.0 以上，JVM 会根据系统配置自行设置，而无须再设置此值。
- -XX:ParallelGCThreads：并行收集器的线程数，值最好配置与处理器数目相等，同样适用于 CMS。
- -XX:+UseParallelOldGC：年老代垃圾收集方式为并行收集（Parallel Compacting）。
- -XX:MaxGCPauseMillis：每次年轻代垃圾回收的最长时间（最大暂停时间），如果无法满足此时间，JVM 会自动调整年轻代大小，以满足此值。
- -XX:+ScavengeBeforeFullGC：Full GC 前调用 YGC，默认是 true。

实例如下：

```
JAVA_OPTS="-Xms4g  -Xmx4g  -Xmn1024m  -XX:PermSize=320M  -XX:MaxPermSize=320m
-XX:SurvivorRatio=6"
```

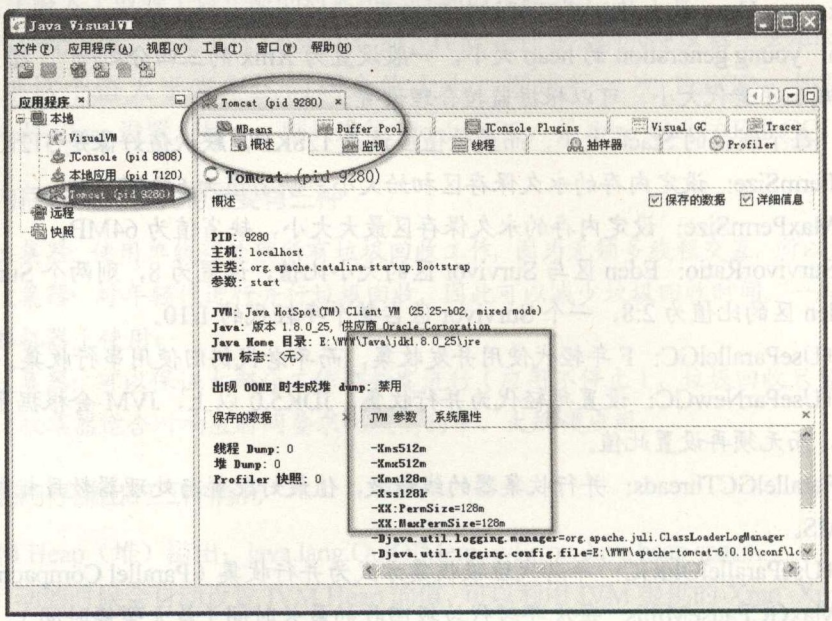
9.4 可视化监控工具的使用

9.4.1 VisualVM 的使用

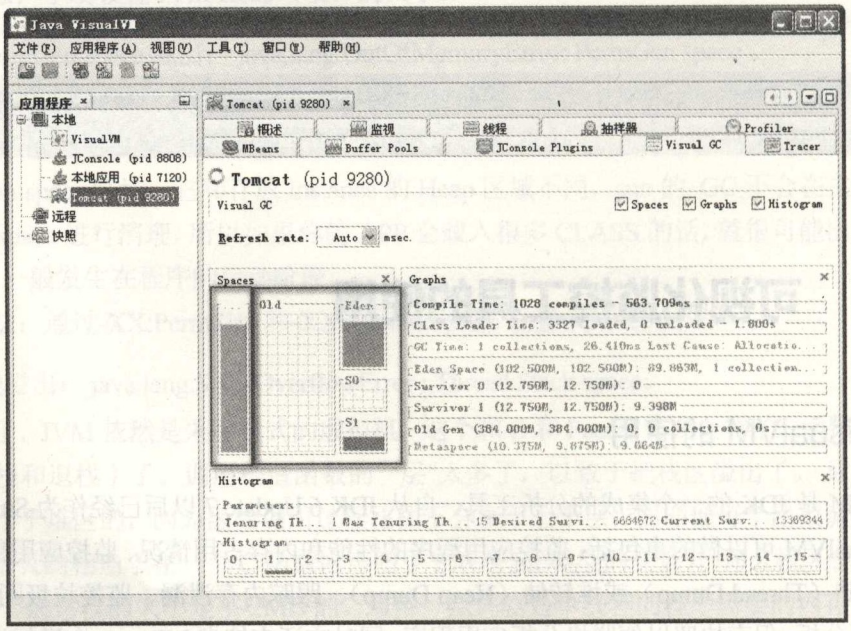
VisualVM 是 JDK 的一个集成的分析工具，自从 JDK 6 Update 7 以后已经作为 Sun 的 JDK 的一部分。VisualVM 可以做的事包括：监控应用程序的性能和内存占用情况、监控应用程序的线程、进行线程转储（Thread Dump）或堆转储（Heap Dump）、跟踪内存泄漏、监控垃圾回收器、执行内存和 CPU 分析，保存快照以便脱机分析应用程序；同时它还支持在 MBeans 上进行浏览和操作。尽管 VisualVM 自身要在 JDK6 以上的运行，但是 JDK1.4 以上版本的程序都能被它监控。

JDK1.6.07以上的版本，在$JAVA_HOME/bin目录下，单击jvisualvm.exe图标就可以启动VisualVM。

（1）单击 jvisualvm 启动 VisualVM，会自动找到我们本地启动的 Tomcat，可以看到我们设置的一些JVM 参数，如下图所示。

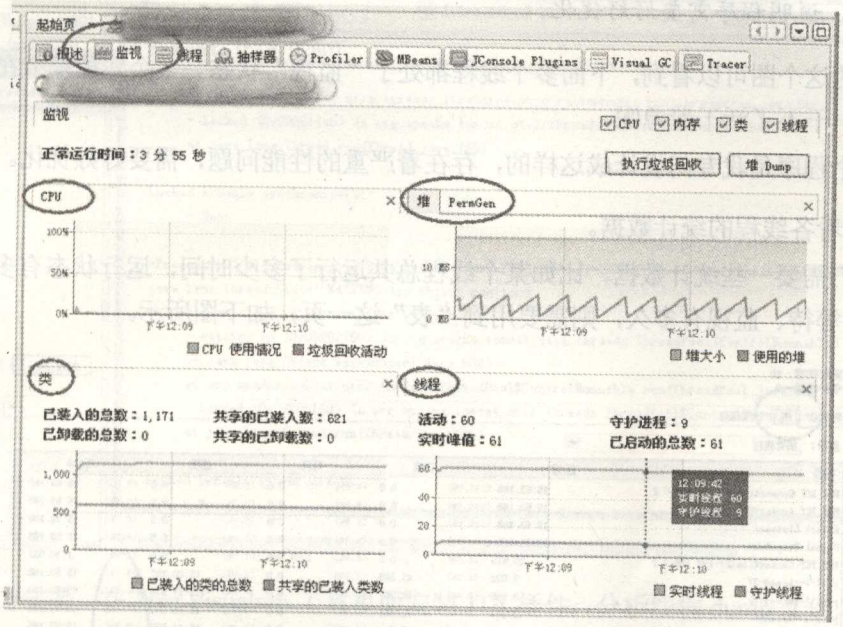


（2）垃圾回收是我们不能忽略的一个地方。我们可以通过“Visual GC”页，查看到非常详细的垃圾回收情况，如下图所示。



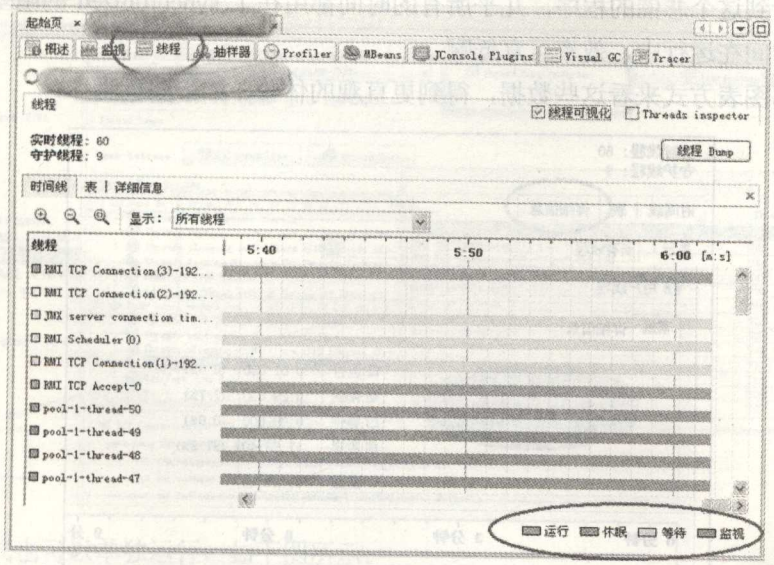
通过这个图我们可以看到我们的永久代分配的有点不太够了，老年代很充足，年轻代也有点不够，会导致后面回收过于频繁。

（3）查看 CPU、堆、类、线程的统计数据，如下图所示。



（4）接下来我们看看如何重点分析和查看线程。

查看各线程运行情况这个是重点，我们需要知道各线程的运行情况，特别是否被 synchronized 阻塞了。如下图所示。



注意上图右下角，有 4 个状态说明，分别是：

- 运行（Running）：我们最喜欢的状态。说明该线程正在执行代码，没有问题。
- 休眠（Sleeping）：调用了 Thread.sleep 后的状态，说明线程正停在某个 Thread.sleep 处。
- 等待（Wait）：手动调用了 wait 方法，或者某些 IO 操作，在阻塞中等待数据。
- 监视（Monitor）：这里就是我想找的问题了。它表示线程想执行一段 synchronized 中的代

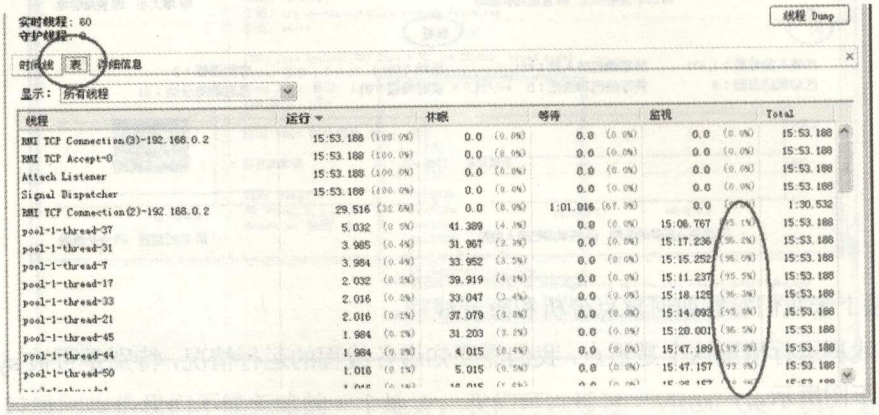
码，但是发现已经有其他线程正在执行，自己被 block 了，只能无奈地等待。如果这种状态多，说明程序需要好好优化。

从上面的这个图可以看到，下面多个线程都处于“监视”状态。多个线程都卡在了独木桥的一头过不去，干不了活干着急呢。

当然这个程序是我专门设计成这样的，存在着严重的性能问题，需要好好优化。

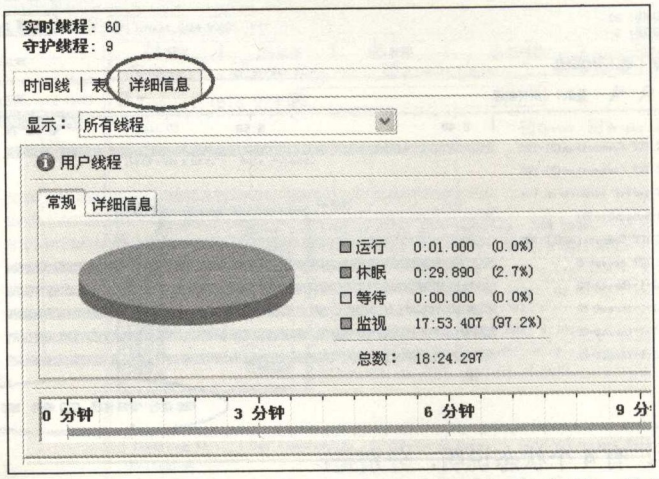
（5）查看各线程的统计数据。

如果我们需要一些统计数据，比如某个线程总共运行了多少时间，运行状态有多久（或百分比），休眠、等待、监视有多久，则需要用到“表”这一页，如下图所示。



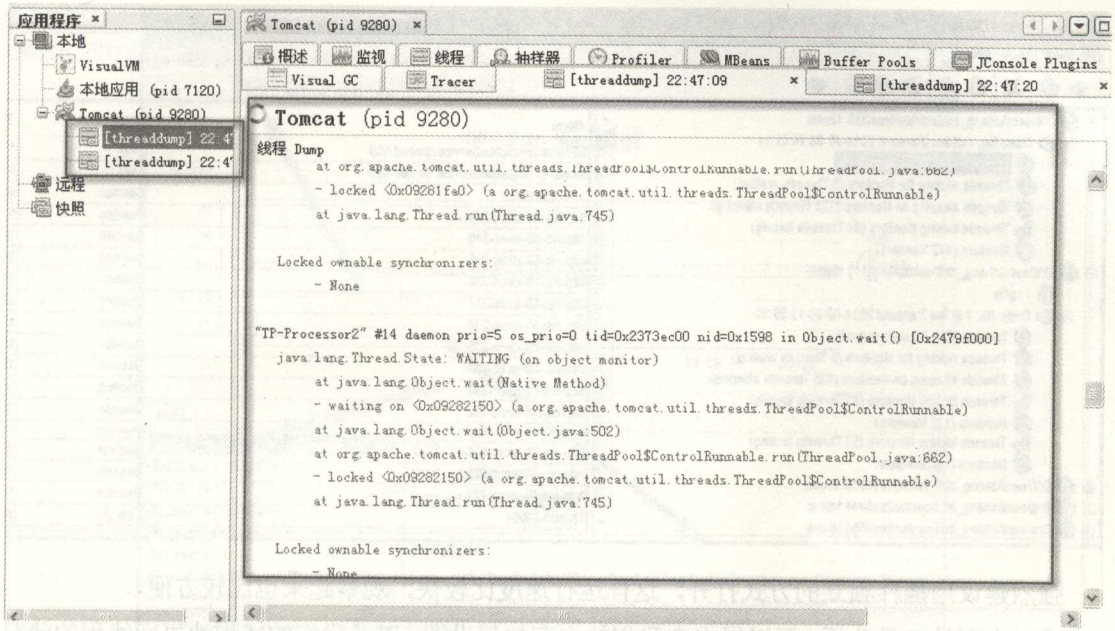
从中可以看到这个悲催的程序，几乎所有的时间都用在了 synchronized 的阻塞上了。只有百分之零点几的时间在运行中，效率可真低啊。

还可以使用图表方式来看这些数据，得到更直观的体验，如下图所示：



（6）有的时候我们需要生产线程 Dump 进行那个时刻的线程分析，我们可以在操作窗口中，点击线程 Dump 即可生产 Dump 文件。

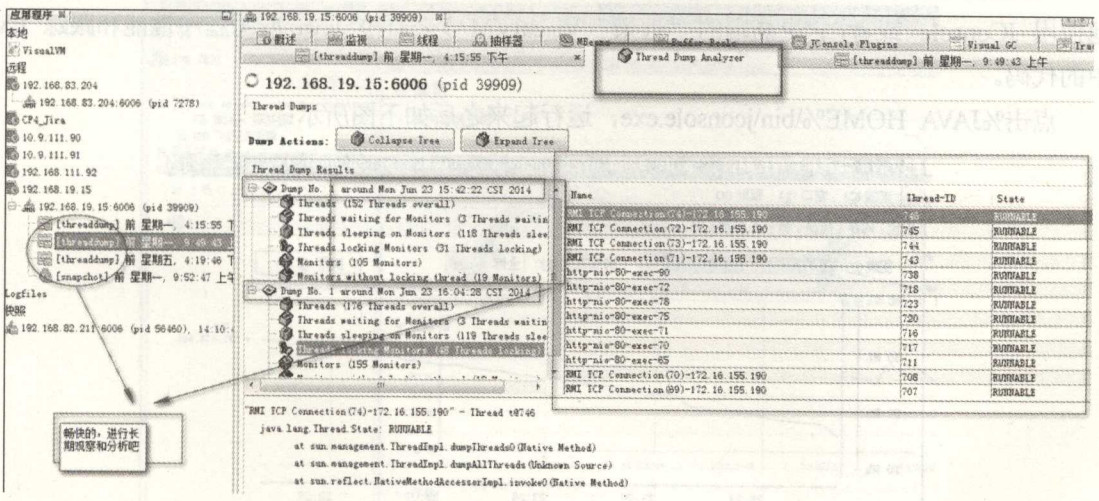
一种方式是通过 threaddump 直接查看，如下图所示：



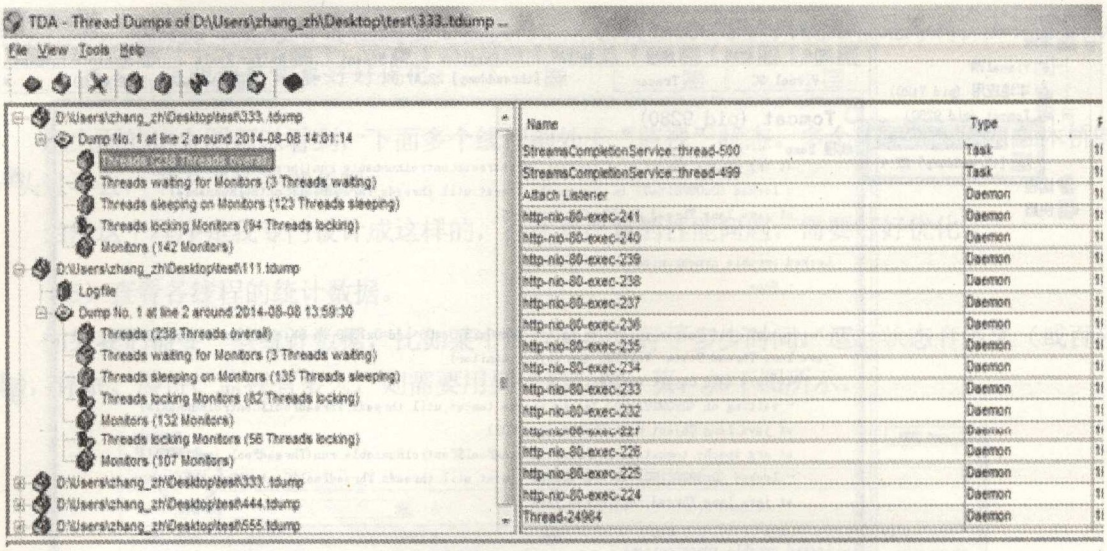
第二种方式是，在 Java Visualvm 工具里面安装 JTA 插件，分析线程 dump 文件。注意，正常阶段的 dump 文件与非正常时期的 Dump 文件进行比较更容易分析出问题。下载地址如下：

```
https://java.net/projects/tda/downloads/directory/visualvm
```

安装好后如下图所示：



第三种方式也可以单独运行，如下图所示：



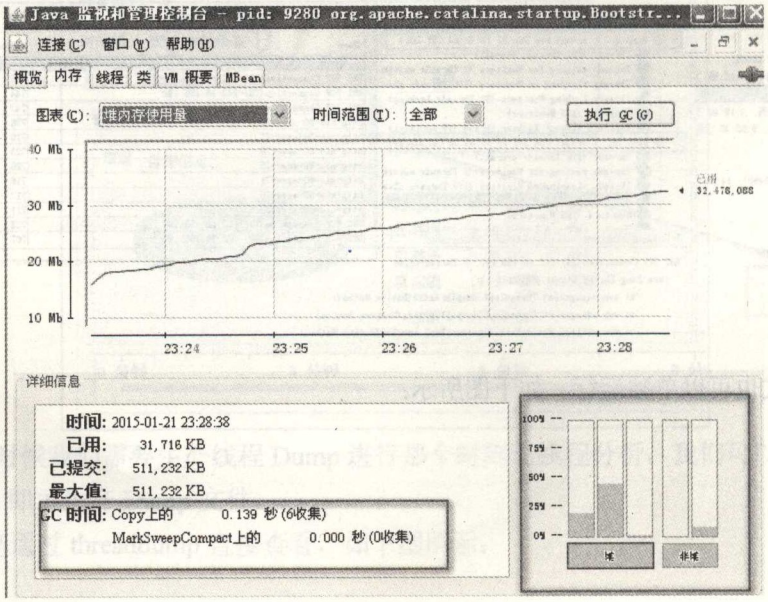
强烈建议用插件独立的方式打开，这样运行速度比较快，观察起来也比较方便。

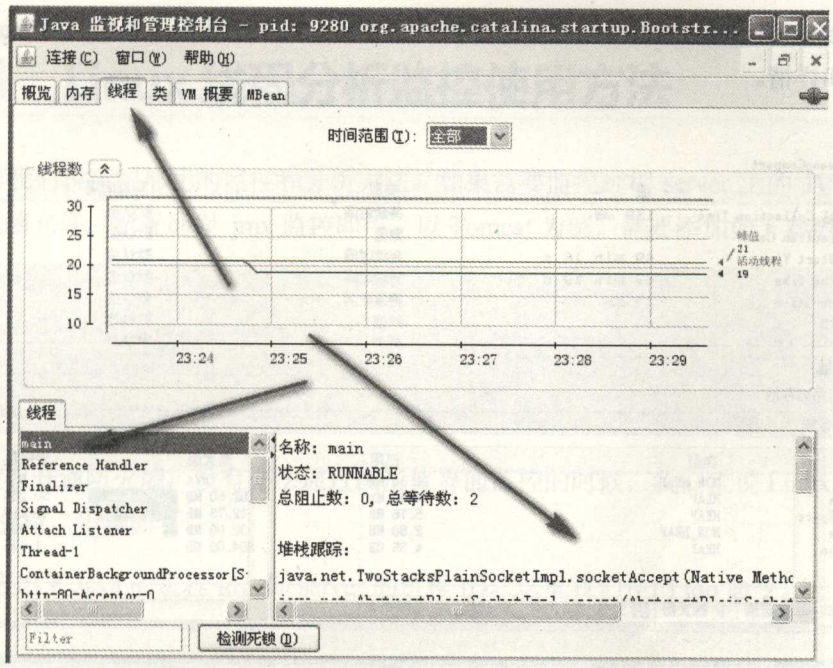
通过上面的工具分析，可以得出来我们的内存如何设置，以及我们的线程池里面线程的状况。结合我们的参数慢慢体会，就会很容易理解 JVM 的运行机制和原理了。

9.4.2 JConsole 的使用

JConsole 是一个内置 Java 性能分析器，可以从命令行或在 GUI shell 中运行。您可以轻松地使用 JConsole（或者，它更高端的“近亲” VisualVM ）来监控 Java 应用程序性能和跟踪 Java 中的代码。

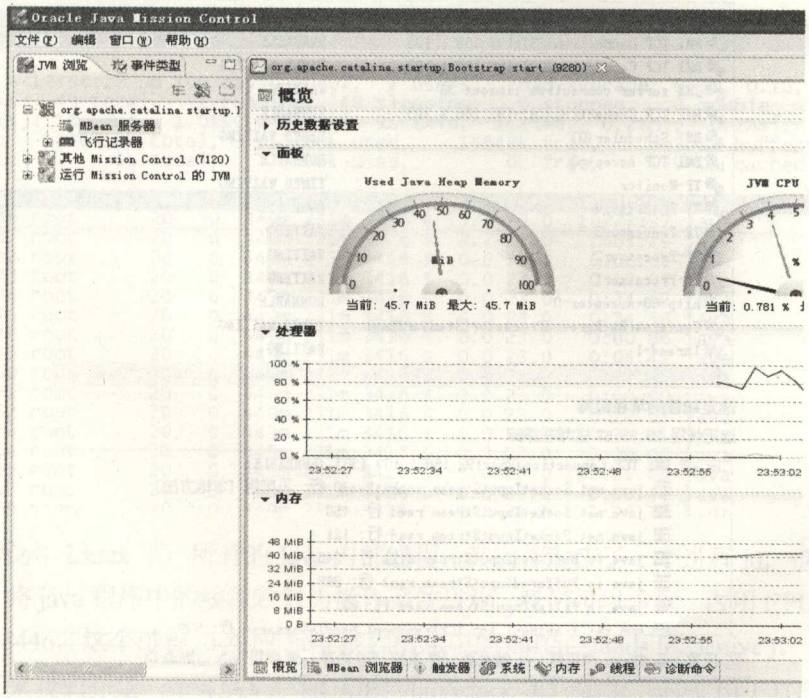
点击%JAVA_HOME%\bin\jconsole.exe，运行起来之后如下图所示：





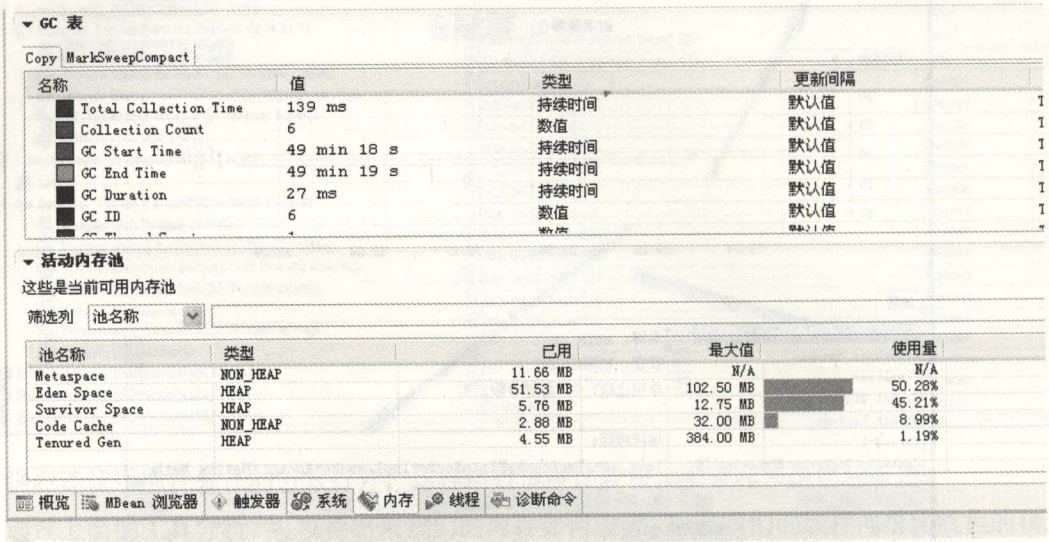
9.4.3　Oracle Java Mission Control

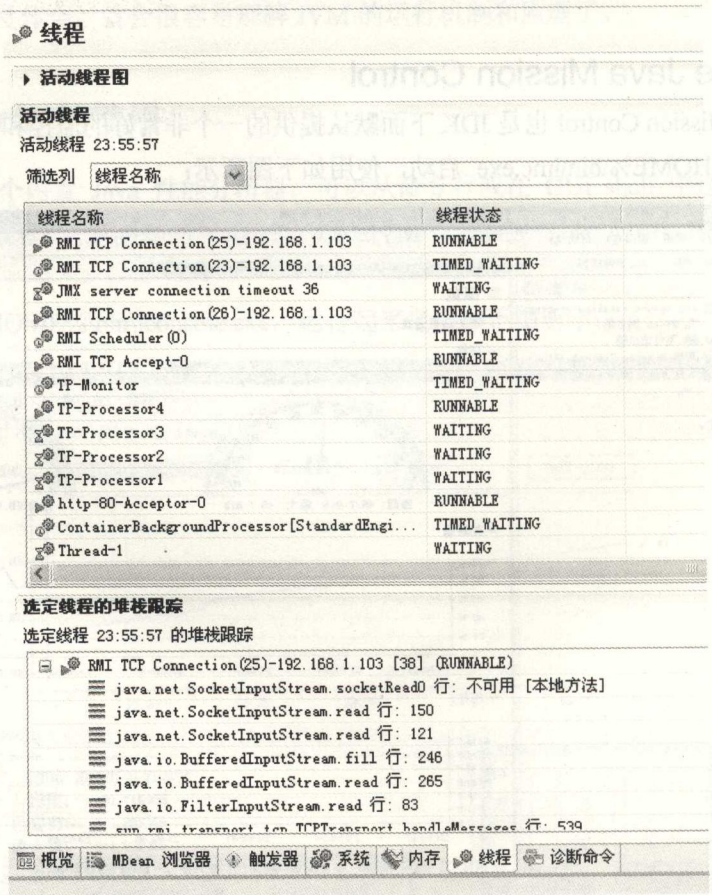
Oracle Java Mission Control 也是 JDK 下面默认提供的一个非常好的监控和分析工具。点击%JAVA_HOME%\bin\jmc.exe 启动，使用如下图所示：



也可以打开飞行模式，分析某一时刻的状态信息。

Oracle Java Mission Control 的功能也是相当的丰富，内存、线程、系统、MBean、配置参数等都有监控和详情。





9.5 Linux 线程分析监控使用方法

上一节我们说了图形化的监控和分析方法，如果需要监控远程 server 上的 JVM 的话，如果端口和防火墙允许，配置远程 jmx 监控即可。以 Tomcat 为例，需要添加如下参数即可：

```
JAVA_OPTS='$JAVA_OPTS
-Djava.rmi.server.hostname=192.168.1.8
-Dcom.sun.management.jmxremote.port=8088
-Dcom.sun.management.jmxremote.ssl=false'
```

如果没有开通防火墙，没有办法通过图形化界面监控的时候，我们说说 Linux 下面如何进行线程 demp 分析，步骤如下。

（1）查出来 Java 的进程 Id，即 server 的进程 ID。这里我们使用 top 命令。

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 8442 root      20   0  449m 231m 3416 S 18.3 23.0  12545:28 java
```

（2）如上图所示，可以看到 Java 的进程 ID 是 8442，接下来用 top 命令单独对这个进程中的所有线程作监视。

我们利用#top -p [进程 ID] -H 这条命令，执行结果如下图所示：

```
[root@ebs-27380 ~]# top -p 8442 -H
```

```
top - 12:54:28 up 100 days, 20:07,  2 users,  load average: 0.39, 0.18, 0.11
Tasks:  49 total,   1 running,  48 sleeping,   0 stopped,   0 zombie
Cpu(s): 16.3%us,  1.0%sy,  0.0%ni, 81.1%id,  1.3%wa,  0.2%hi,  0.2%si,  0.0%st
Mem:   1030916k total,  1015268k used,    15648k free,    69308k buffers
Swap:        0k total,        0k used,        0k free,    65940k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 8446 root      20   0  449m 231m 3416 R 13.3 23.0  10777:37 java
 8447 root      20   0  449m 231m 3416 S  3.7 23.0   1663:19 java
 8442 root      20   0  449m 231m 3416 S  0.0 23.0   0:00.00 java
 8443 root      20   0  449m 231m 3416 S  0.0 23.0   0:12.05 java
 8444 root      20   0  449m 231m 3416 S  0.0 23.0   0:24.42 java
 8445 root      20   0  449m 231m 3416 S  0.0 23.0   0:24.58 java
 8448 root      20   0  449m 231m 3416 S  0.0 23.0   0:00.86 java
 8449 root      20   0  449m 231m 3416 S  0.0 23.0   0:04.47 java
 8450 root      20   0  449m 231m 3416 S  0.0 23.0   4:27.85 java
 8451 root      20   0  449m 231m 3416 S  0.0 23.0   0:00.00 java
 8452 root      20   0  449m 231m 3416 S  0.0 23.0   0:12.03 java
 8453 root      20   0  449m 231m 3416 S  0.0 23.0   0:23.41 java
 8454 root      20   0  449m 231m 3416 S  0.0 23.0   0:00.00 java
 8455 root      20   0  449m 231m 3416 S  0.0 23.0 127:29.48 java
 8457 root      20   0  449m 231m 3416 S  0.0 23.0   0:00.44 java
 8466 root      20   0  449m 231m 3416 S  0.0 23.0   3:36.79 java
```

如上图所示，Linux 下，所有的 Java 内部线程，其实都对应了一个进程 id，也就是说，Linux 上的 sun jvm 将 java 程序中的线程映射为操作系统进程。我们可以看到，占用 CPU 资源最高的那个进程 id 是 8446，这个进程 id 对应 Java 线程信息中的'nid'('n' stands for 'native')。

（3）接下来我们介绍一个%JAVA_HOME%/bin/jstack 执行脚本。jstack 用于打印出给定的 Java 进程 ID 或 core file 或远程调试服务的 Java 堆栈信息。我们利用#jstack pid 生产 java 的线程 dump

信息，如下图所示。

```
[root@ebs-27380 ~]# jstack 8442> ./8442_dump.txt
```

```
[root@ebs-27380 ~]# cat 8442_dump.txt
2015-01-24 12:59:43
Full thread dump Java HotSpot(TM) Server VM (11.2-b01 mixed mode):

"Attach Listener" daemon prio=10 tid=0x9cc3c000 nid=0x4162 waiting on condition [0x
   java.lang.Thread.State: RUNNABLE

"Java2D Disposer" daemon prio=10 tid=0x9e09dc00 nid=0x5723 in Object.wait() [0x9cbf
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xa85cfb10> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
        - locked <0xa85cfb10> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
        at sun.java2d.Disposer.run(Disposer.java:125)
        at java.lang.Thread.run(Thread.java:619)

"http-8010-19" daemon prio=10 tid=0x9f8aac00 nid=0x3aa1 in Object.wait() [0x9cdec00
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
```

（4）然后我们将第二步找到的线程 ID：8446 转换成 16 进制 20fe，然后到 8442_dump.txt 中即可找到这条线程的信息：

```
"Gang worker#0 (Parallel GC Threads)" prio=10 tid=0xb7409000 nid=0x20fc runnable
"Gang worker#1 (Parallel GC Threads)" prio=10 tid=0xb740a800 nid=0x20fd runnable
"Concurrent Mark-Sweep GC Thread" prio=10 tid=0xb7458400 nid=0x20fe runnable
```

可以看到它是一条垃圾回收器里面的并发线程。

Concurrent Mark-Sweep GC Thread 并发标记清除垃圾回收器（就是通常所说的 CMS GC）线程，该线程主要针对于老年代垃圾回收。ps：启用该垃圾回收器，需要在 jvm 启动参数中加上：

```
-XX:+UseConcMarkSweepGC
```

（5）当然了我们也可以将 dump 的 txt 文件从服务器上面拿下来，再使用我们上面讲的图形化工具进行分析。

（6）一般一个线程的 dump 主要包含如下信息：

- 线程名称：Concurrent Mark-Sweep GC Thread。
- 线程类型：daemon。
- 优先级：prio=10，默认是 5。
- jvm 线程 id：tid=0xb7458400，jvm 内部线程的唯一标识（通过 java.lang.Thread.getId()获取，通常用自增方式实现）。
- 对应系统线程 id（NativeThread ID）：nid=0x20fe，和 top 命令查看的线程 pid 对应，不过一个是 10 进制，一个是 16 进制（通过命令 top -H -p pid，可以查看该进程的所有线程信息）。
- 线程状态：runnable。

下面我们谈谈 Linux 下面如何查看和分析 java 的内存情况。

（1）我们介绍一个%JAVA_HOME%/bin/jstat 执行脚本。Jstat 是 JDK 自带的一个轻量级小工具，全称为 Java Virtual Machine statistics monitoring tool，它位于 Java 的 bin 目录下，主要利用 JVM 内建的指令，对 Java 应用程序的资源和性能进行实时的命令行的监控，包括了对 Heap size 和垃圾回收状况的监控。可见，Jstat 是轻量级的、专门针对 JVM 的工具，非常适用。由于 JVM 内存设置较大，图中百分比变化不太明显。一个极强的监视 VM 内存工具，可以用来监视 VM 内存内的各种堆和非堆的大小及其内存使用量。jstat 工具特别强大，有众多的可选项，详细查看堆内各个部分的使用量，以及加载类的数量。使用时，需加上查看进程的进程 id 和所选参数。详细的语法请看书后面的附录 2 和附录 3。使用和分析例子如下所示：

```
[root@ebs-27380 ~]# jstat -gcutil -h 10 8442 1000
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
70.34   0.00  62.17  29.16  59.92   2090   93.111 1200391 97501.443 97594.554
70.34   0.00  62.17  29.16  59.92   2090   93.111 1200392 97501.561 97594.672
70.34   0.00  62.17  29.16  59.92   2090   93.111 1200392 97501.561 97594.672
70.34   0.00  62.17  29.16  59.92   2090   93.111 1200393 97501.659 97594.770
```

（2）从上面的结果我们可以看得出来，年轻代占了 62.17%，老年代占了 29.16%，永久代占了 59.92%，而 YGC 的次数为 2 090 次，FGC 的次数为 1 200 391 并且还在一直不断地增加，结合上面我们线程的分析来看，确实一直在发生着 FullGC。到这里基本上已经能够看到我们在启动 server 的时候参数设置得不是很正确，导致什么都没做的情况下一直在发生 FullGC，在浪费服务器 CPU。

我们通过 ps -ef|grep java 这个命令找到了正在运行的这个 java Tomcat 的进程，发现了 JVM 设置了如下一些参数。

```
root      8442     1 17  2014 ?        8-17:20:24 /usr/java/jdk1.6.0_12/bin/java
-Xms200M -Xmx200M -Xmn80M
-XX:PermSize=50M
-XX:MaxPermSize=100M
-XX:SurvivorRatio=5  #设置为5,则两个 Survivor 区与一个 Eden 区的比值为2:5
-XX:MaxTenuringThreshold=7  #年轻代交换的次数，该参数只有在串行 GC 时才有效.
-Xnoclassgc  #禁用垃圾回收
-XX:+DisableExplicitGC #关闭 System.gc()
-XX:+UseParNewGC #设置年轻代为并行收集
-XX:+UseConcMarkSweepGC  #使用 CMS 内存收集
-XX:+UseCMSCompactAtFullCollection #在 FULL GC 的时候， 对年老代的压缩
-XX:CMSInitiatingOccupancyFraction=50  #使用 cms 作为垃圾回收使用50%后开始 CMS 收集
-XX:SoftRefLRUPolicyMSPerMB=0 #每兆堆空闲空间中 SoftReference 的存活时间
-XX:+HeapDumpOnOutOfMemoryError #开启当发生 OutOfMemory 的时候生成 dump 文件
-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
```

```
   -classpath /usr/local/server/tomcat/bin/bootstrap.jar -Dcatalina.base=
/usr/local/server/tomcat
   -Dcatalina.home=/usr/local/server/tomcat org.apache.catalina.startup.Bootstrap
start
```

从上面的信息我们可以看得出来，我们为了测试演练，将 -XX:CMSInitiatingOccupancyFraction=50，表示使用 cms 作为垃圾回收使用 50%后开始 CMS 收集，而且堆的最大内存和最小内存（-Xms200M -Xmx200M）以及（年轻代-Xmn80M）都给的很小。所以刚才 jstat 监控的时候，三个代区的百分比才会很大，实际工作中如果这样的百分比是要出大事的。

9.6 Linux 分析监控的运行脚本

下面我们分享一个 Linux 会常用的 sh 脚本，功能是生成 3 个 dump，方便对比分析当前时刻的线程 dump，可以用在 CPU 很高的时候，重启之前执行，也可以用在正在运行的时候执行，总之就是找几个正常的 thread dump 和几个非正常时期的 dump 进行对比分析。内容如下：

```
#!/bin/sh
# dump sh
##先活动一个 tomcat 的运行的进程 ID
java_pid=`ps -ef|grep '^.*java.*tomcat.*$'|grep -v grep|awk '{print $2}'`
echo "java pid is :$java_pid"
##找到 jstack
jstack_path=/opt/jdk/jdk1.7.0_60/bin/jstack
##thread dump 存放路径
logs_path=/tmp/java/thread_dump

##取当前 tomcat 进程里面的 CPU 最高的前15个的线程信息存到临时文件里面
top -p $java_pid -H |head -15 > $logs_path/top_dump_111.txt
##取当前 tomcat 的 thread dump 储存到临时文件里面方便分析
sudo $jstack_path $java_pid > $logs_path/111.dump
echo "create $logs_path/111.dump success, sleep 10s;"
##等待10秒在重复上面的过程，生成3个 dump 方便分析和对比。
sleep 10s

top -p $java_pid -H |head -15 > $logs_path/top_dump_222.txt
sudo $jstack_path $java_pid > $logs_path/222.dump
```

```
echo "create $logs_path/222.dump success, sleep 10s;"
echo "create $logs_path/top_dump_222.txt success, sleep 10s;"
sleep 10s

top -p $java_pid -H |head -15 > $logs_path/top_dump_333.txt
sudo $jstack_path $java_pid > $logs_path/333.dump
echo "create $logs_path/333.dump success, sleep 10s;"
echo "create $logs_path/top_dump_333.txt success, sleep 10s;"
sleep 10s
```

只需要将 top_dump_111.txt 和 111.dump、top_dump_222.txt 和 222.dump、top_dump_333.txt 和 333.dump 对比分析即可。

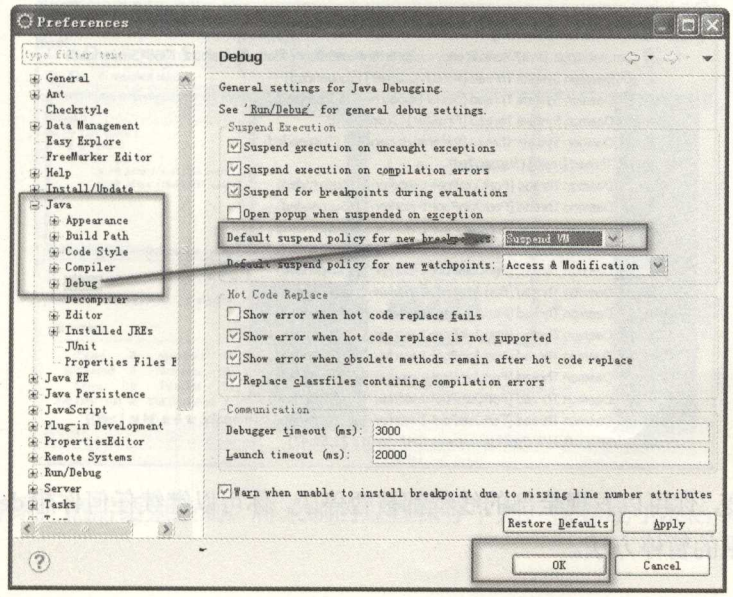
很多时候，我们可以通过工具发现占用 CPU、内存很高的线程的话，就用不到上面的脚本了，但准备好这个脚本有的时候还是有备无患的。

9.7 Eclipse 里面如何调试并发程序

Eclipse 是最普遍的 IDE 之一。它有一个内置调试器（integrated debugger）允许你测试你的应用。默认情况，等你试调并发应用，debugger 找到断点（breakpoint），它只是把有断点（breakpoint）的线程停止，其他的线程都继续他们的运行。

Eclipse 里面有一个设置可以暂停 JVM，而不是暂停当前线程，具体设置如下所述。

步骤 01　选择菜单选项 Window→Preferences。如下图所示。

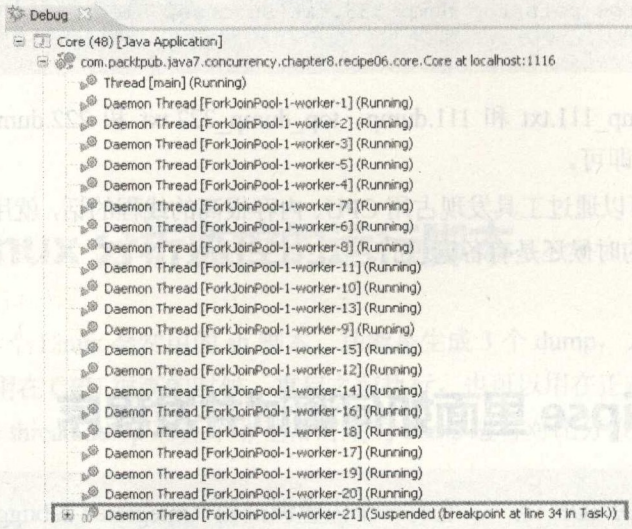


步骤 02 改变 Default suspend policy 的值，为了 new breakpoints 把 Suspend Thread 改成 Suspend VM（在截图中用红色标记了）。

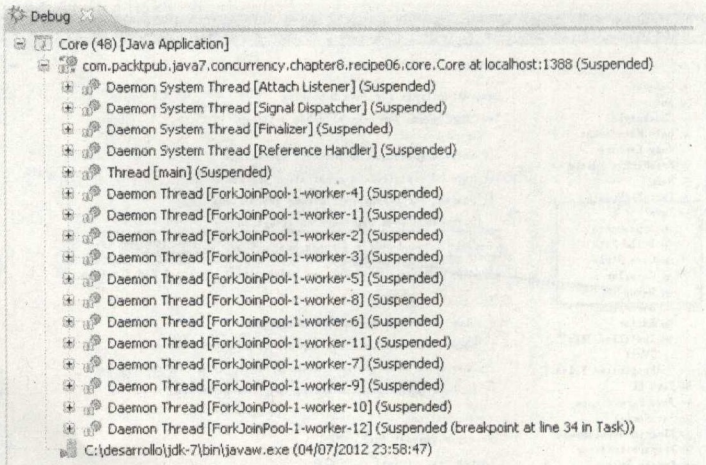
步骤 03 单击 OK 按钮来确定修改即可。

实际使用对比如下：

默认情况，当你用 Eclipse 调试并发 Java 应用，调试过程会寻找断点，它只暂停最先碰到这个断点的线程，其他线程将继续他们的运行。以下截图展示了例子的情况：



你可以发现只有 worker-21 被暂停了（在截图中用红色标记），而其他线程还在继续运行。但是，如果你改变 Default suspend policy 到 Suspend VM，来获得新的 breakpoints，全部线程暂停他们的运行，当你正在 debugging 并发应用，然后试调过程碰到了断点。以下截图展示了例子的情况：



有了这些改变，你可以发现全部的线程都被暂停了。你可以继续任何你想 debugging 的线程。选择最适合你需求的暂停方法。

9.8 如何通过压力测试来测试服务器的抗压能力

1. Badboy

Badboy 也是一个强大的测试工具，被设计用于测试和开发复杂的动态应用。Badboy 功能丰富（包括一个捕获/重播接口，强大的压力测试支持，详细的报告、图形）使得测试和开发更加容易。

Badboy，有人称为“坏孩子”的意思，其实，它是专门用来给项目找茬来的。Badboy 是用来录制操作过程的，然后另存为 JMete 脚本，也就是说它录制的结果是被 JMeter 做并发测试的素材使用。有关这个录制过程比较简单，读者可以上网查看相关文档学习。

Badboy 首页：http://www.badboy.com.au/.

2. JMeter

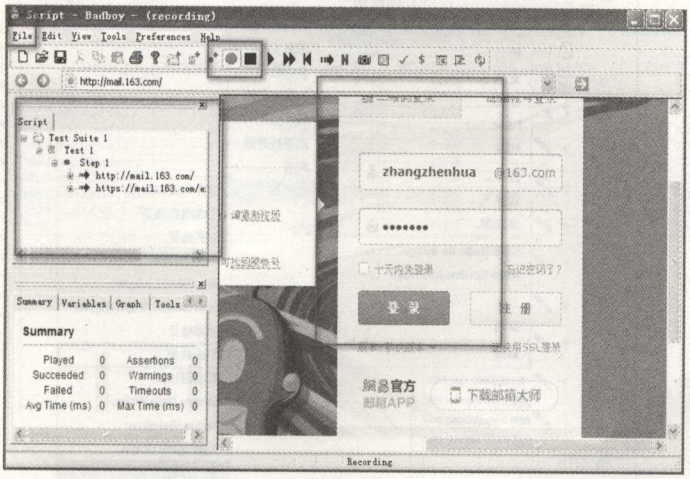
JMeter 是 Apache 下的一个完全基于 JAVA 开发的测试工具，可以很方便地用来进行并发测试。

Apache JMeter 是一个专门为运行和服务器装载测试而设计的、100%的纯 Java 桌面运行程序。原先它是为 Web/HTTP 测试而设计的，但是它已经扩展以支持各种各样的测试模块。它和用于 HTTP 和 SQL 数据库（使用 JDBC）的模块一起运送。它可以用来测试静止资料库或者活动资料库中的服务器的运行情况，可以用来模拟对服务器或者网络系统加以重负荷以测试它的抵抗力，或者用来分析不同负荷类型下的所有运行情况。它也提供了一个可替换的界面用来定制数据显示，测试同步及测试的创建和执行。JMeter 首页链接如下：

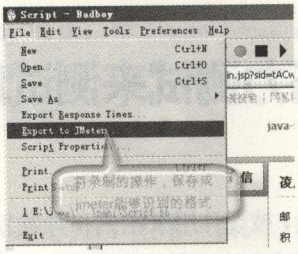
```
http://jakarta.apache.org/JMeter/
```

3. 两者接合的使用过程

（1）打开 badboy 进行登录的录制工作，如下图所示。

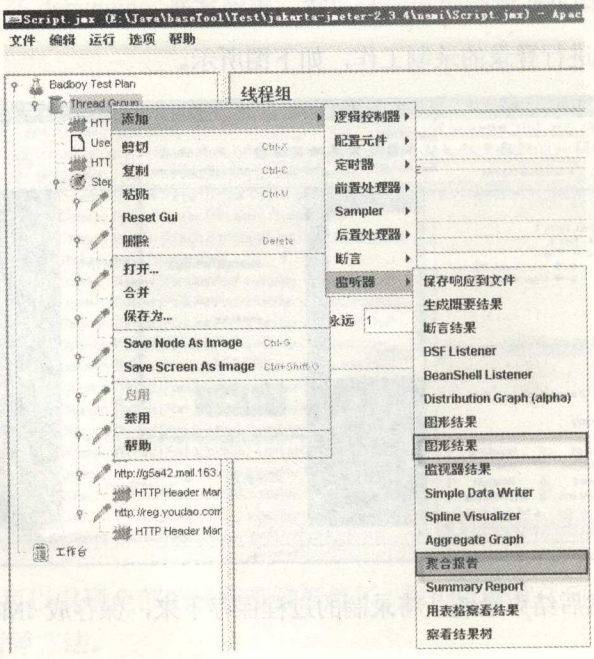


（2）单击登录，然后结束录制，将录制的过程保存下来，保存成 JMeter 能够使用的文件格式 Script.jmx：

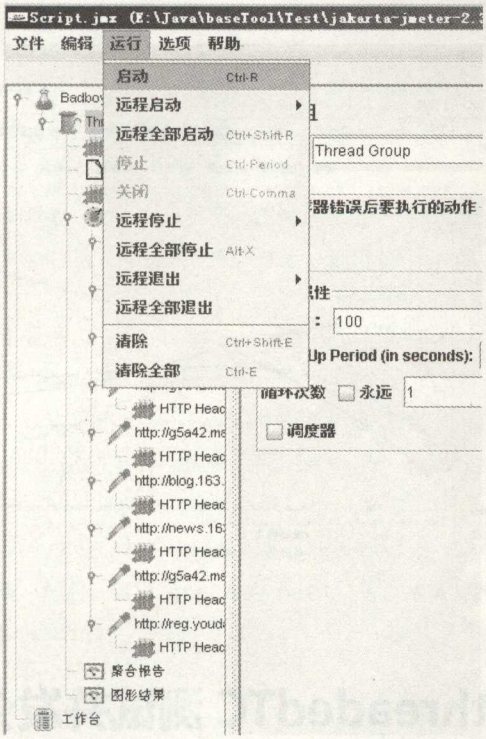


（3）运行 Apache JMeter，单击文件❼打开，然后选择刚才保存的录制文件 Script.jmx，设置模拟并发的线程数量，如下图所示。

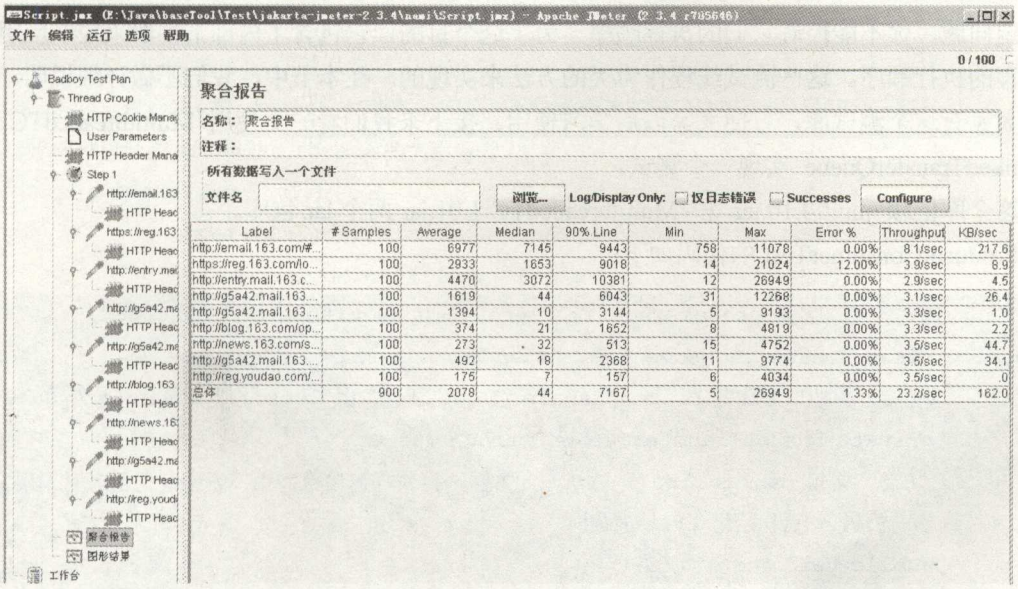
（4）添加感兴趣的监听类型，如下图所示。

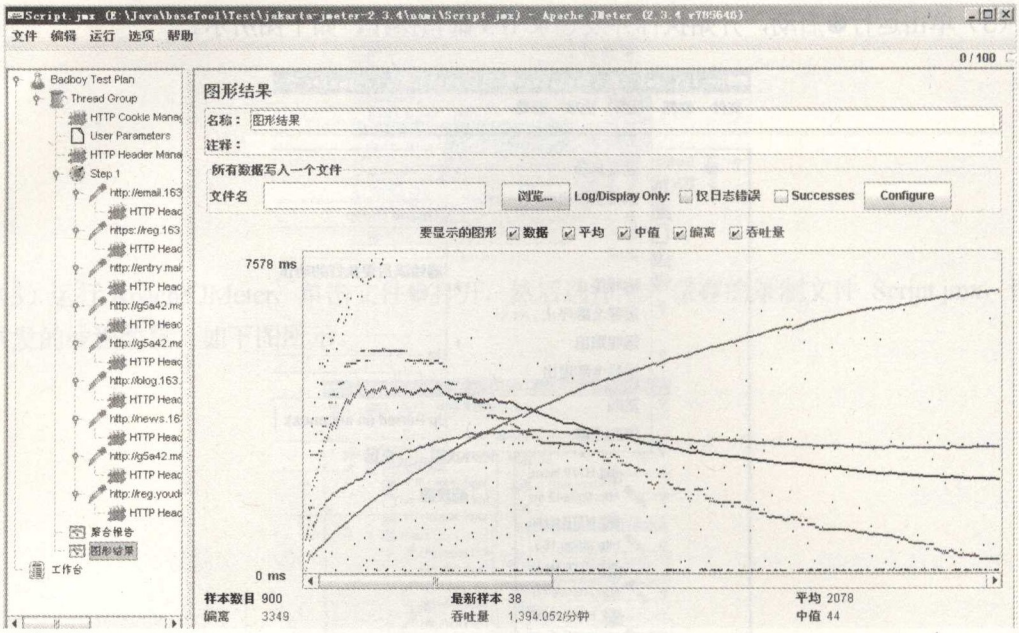


（5）单击运行❼启动，开始执行并发登录 163 邮箱操作，如下图所示。



（6）分析结果，如下图所示。





9.9 MultithreadedTC 测试并发介绍

MultithreadedTC 是一个 Java 库，用来测试并发应用。它的主要目的是为了解决并发应用的不确定问题，你不能控制它们的执行顺序。为了这个目的，它包含了内部节拍器来控制应用的不同线程的执行顺序。这些测试线程作为类的方法来实现的。在本节中，我们不做详细说明，只是告诉大家有这个测试库，方便大家以后学习使用。接下来我们看一个使用 MultithreadedTC 库来为 LinkedTransferQueue 实现一个测试。

这个库依赖 junit-4.10.jar 和 MultithreadedTC-1.01.jar 两个 jar 包库。

ProducerConsumerTest 的代码如下：

```
// 1.创建一个类，名为 ProducerConsumerTest，扩展 MultithreadedTestCase 类。
public class ProducerConsumerTest extends MultithreadedTestCase {
        // 2. 声明一个私有 LinkedTransferQueue 属性，用 String 类为参数，名为 queue。
        private LinkedTransferQueue<String> queue;
        // 3. 实现 initialize() 方法。此方法不接收任何参数，也不返回任何值。它调用父类的
initialize() 方法，然后初始化 queue 属性。
        public void initialize() {
                super.initialize();
                queue = new LinkedTransferQueue <String>();
                System. out.printf("Test: The test has been initialized\n" );
        }
```

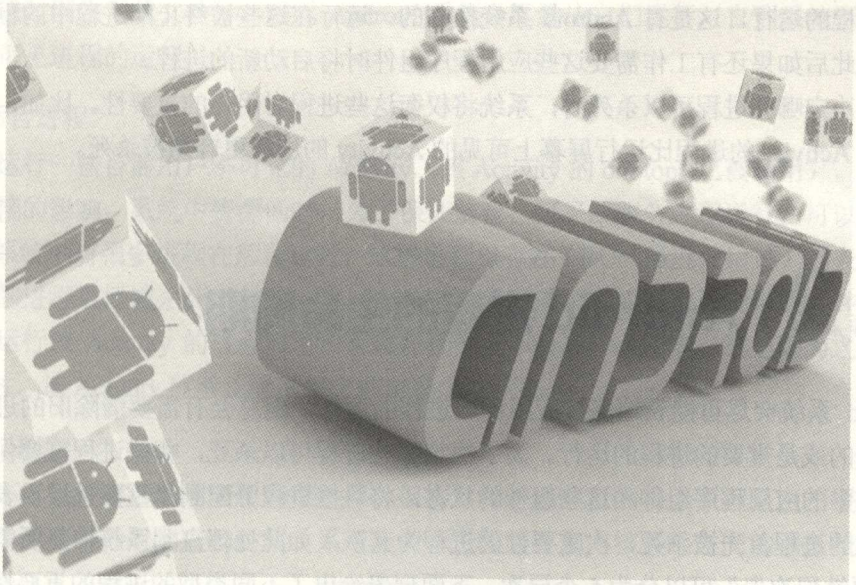
```
        // 4. 实现 thread1() 方法。它将实现的逻辑是第一个 consumer。调用 queue 的 take()
方法，然后把返回值写入操控台。
        public void thread1() throws InterruptedException {
            String ret = queue.take();
            System. out.printf("Thread 1: %s\n" , ret);
        }
        //5. 实现 thread2()方法。它将实现的逻辑是第二个 consumer。首先，使用 waitForTick()
方法，一直等到第一个线程在 take()方法中进入休眠。然后，调用 queue 的 take()方法，并把返回值写入操
控台。
        public void thread2() throws InterruptedException {
            waitForTick(1);
            String ret = queue.take();
            System. out.printf("Thread 2: %s\n" , ret);
        }
    // 6. 实现 thread3() 方法。它将实现的逻辑是 producer。 首先，使用 waitForTick()两次
一直等到2个 consumers 被阻塞。然后，调用 queue 的 put() 方法插入2个 String 到 queue 中。
        public void thread3() {
            waitForTick(1);
            waitForTick(2);
            queue.put("Event 1" );
            queue.put("Event 2" );
            System. out.printf("Thread 3: Inserted two elements\n" );
        }
    // 7. 最后，实现 finish() 方法。写信息到操控台表明测试结束执行。使用 assertEquals()方
法检查2个事件已经被 consumed (queue 的大小为0)。
        public void finish() {
            super.finish();
            System. out.printf("Test: End\n" );
            assertEquals( true, queue .size() == 0);
            System. out.printf("Test: Result: The queue is empty\n" );
        }
}

// 8. 创建例子的主类，通过创建一个类，名为 Main 并添加 main()方法。
public class Main {
        public static void main(String[] args) throws Throwable {
            // 9. 创建 ProducerConsumerTest 对象，名为 test。
            ProducerConsumerTest test = new ProducerConsumerTest();
            // 10. 使用 TestFramework 类的 runOnce()方法来执行测试。
```

```
            System.out.printf("Main: Starting the test\n");
            TestFramework.runOnce(test);
            System.out.printf("Main: The test has finished\n");
        }
    }
```

上面实例中使用 MultithreadedTC 库为 LinkedTransferQueue 类实现了一个测试。你可以使用这个库和它的节拍器为任何并发应用或者类实现测试。在例子中，你实现经典的 producer/consumer 问题，2 个 consumers 和一个 producer。你想要测试的是，在 buffer 里的第一个介绍的 String 对象会被第一个 consumer 消耗，和在 buffer 里第二个介绍的 String 对象会被第二个到达的 consumer 消耗。详细的分析我们就不在这里说明了，大家知道有这个东西即可，实际工作中用到这个的相对来说比较少一些。

第 10 章
Android中线程的应用

很多事情不是看到希望了才去坚持，而是坚持了才会看到希望。学习也是如此。



其实进程线程到哪里都是差不多的，让我们一起认识一下 Android 中的进程与线程。

10.1 Android 进程基本知识

大家都知道 Android 是基于 Linux 平台的开源手机操作系统，该平台由操作系统、中间件、用户界面和应用软件组成。所以呢？很多运行原理和机制都和 Linux 上的线程和进程的概念差不多。当一个 Android 启动一个应用程序组件时，如果该应用没有正在运行的其他程序组件，那么 Android 系统将为这个应用创建一个新进程（包含一个线程）用于运行应用。缺省情况下，一个应用的所有组件（Activity，Service 等）运行在同一个进程和线程中（称为“主”线程）。如果在启动一个应用程序组件时，这个应用已经有进程在运行（因为有应用的其他组件存在），那么这个应用程序组件将使用同一进程和线程运行。当然你可以使用不同进程来运行不同的组件，或者

在进程中创建新的线程。

缺省情况下，应用的所有组件都运行在同一个进程中，而且应用不应该改变这个传统。然而，如果你发现你需要控制某个组件运行在哪个进程中，你可以通过应用程序清单来配置。

在应用程序清单文件中，每个类型的应用程序组件<activity>、<service>、<receiver>和<provider>都支持 android:process 属性，这个属性用来指明该程序组件运行的进程。你可以为应用程序组件设置这个属性，以使每个组件运行在不同的进程中，或者某几个组件使用同一进程。你也可以通过设置 android:process 使得不同应用中的组件运行在同一个进程中，前提是这些应用使用同一个 Linux 用户名并且使用同一个证书签名。<Application>元素也支持 android: process 属性，用来为应用程序的所有组件设置默认的进程。

Android 系统中系统资源过低，而且有需要为用户立即提供服务的进程需要启动时，可能会终止某些进程的运行，这是有 Android 系统控制的。运行在这些被终止的进程中的程序组件将逐个被销毁。此后如果还有工作需要这些应用程序组件时将启动新的进程。

系统中决定哪些进程可以杀死时，系统将权衡这些进程对用户的重要性。比如，对于那些运行不可见的 Activity 的进程比运行屏幕上可见的 Activity 的进程更容易被杀死。

10.2 Android 进程的生命周期

Android 系统会尽可能长地保持应用程序进程的运行，但总会有需要清除旧的进程来释放资源以满足新的或是重要的进程的运行。为了决定哪些进程可以杀死，哪些进程需要保留，系统根据运行在其中的应用程序组件和这些组件的状态，将这些进程分配到“重要性层次表”中。具有最低重要性的进程首先被杀死，次重要性的进程为其次，如此处理直到系统恢复所需的资源。

“重要性层次表”可以分为 5 个层次，下面列表给出了不同类型的进程的重要性等级（最重要的排在前面，也就是最晚被杀掉的）：

（1）前台进程

这种进程是当前用户正在玩的应用，正在看的应用，与用户正在交互的应用。一个进程被认为是前台进程需满足下面条件之一：

- 本进程中有 Activity 是与当前的用户有交互的（即，此应用中的 Activity 的 onResume() 被调用）。
- 本进程中有 Service 和当前与用户有交互 Activity 有绑定的关系。
- 本进程中有在前台运行的 Service(即，Service 调用过 startForeground())。
- 本进程中有 Service 正在执行某个生命周期回调函数（如 onCreate()、onStart()或 onDestroy()）。
- 本进程中有 Service 正在执行某个生命周期回调函数（如 onCreate()、onStart()或 onDestroy()）。

（2）可见进程

这种进程虽然不含有任何在前台运行的组件，但会影响当前显示给用户屏幕上的内容，一个进程中满足下面两个条件之一时被认为是个可见进程:

- 本进程含有一个虽然不在前台但却部分可见的 Activity(该 Activity 的 onPause()被调用)。可能发生的情形是前台 Activity 显示一对话框，此时之前的 Activity 变为部分可见。
- 本进程含有绑定到可见 Activity 的 Service。

（3）服务进程

该进程运行了某个使用 startService()启动的 Service，但不属于以上两种情况。尽管此服务进程不直接和用户可以看到的任何部分有关联，但它会运行一些用户关心的事情（比如，在后台播放音乐或者通过网络下载文件）。因此 Android 系统会尽量让它们运行，直到系统资源低到无法满足前台和可见进程的运行。

（4）后台进程

该进程运行一些目前用户不可见的 Activity（该 Activity 的 onStop()已被调用），该进程对用户体验无直接的影响，系统中资源低时，为了保证前台、可见或服务进程运行，可以随时杀死该进程。通常系统中有很多进程在后台运行，这些进程保存在 LRU（最近使用过）列表中以保证用户最后看到的进程最后被杀死。如果一个 Activity 正确实现了它的生命周期函数，并保存了它的状态。杀死运行该 Activity 的进程对用户来说在视觉上不会有什么影响，这是因为之后用户回到该 Activity 时，该 Activity 能够正确恢复之前屏幕上的状态。

（5）空进程

该进程不运行任何活动的应用程序组件。保持这种进程运行的唯一原因是由于缓存，以缩短下次运行某个程序组件时的启动时间。系统会为了进程缓存和内核缓存之间的平衡经常会清除空进程。

Android 系统会根据进程中当前活动的程序组件的重要性，尽可能高地给该进程评级。比如，如果一个进程中同时有一个 Service 和一个可见的 Activity 在运行，该进程将被定级为可见进程而不是服务进程（因为可见进程的优先级高于服务进程）。也就说在该进程，同时满足多个优先级进程的时候，会以该进程最高的优先级作为该进程的优先级。

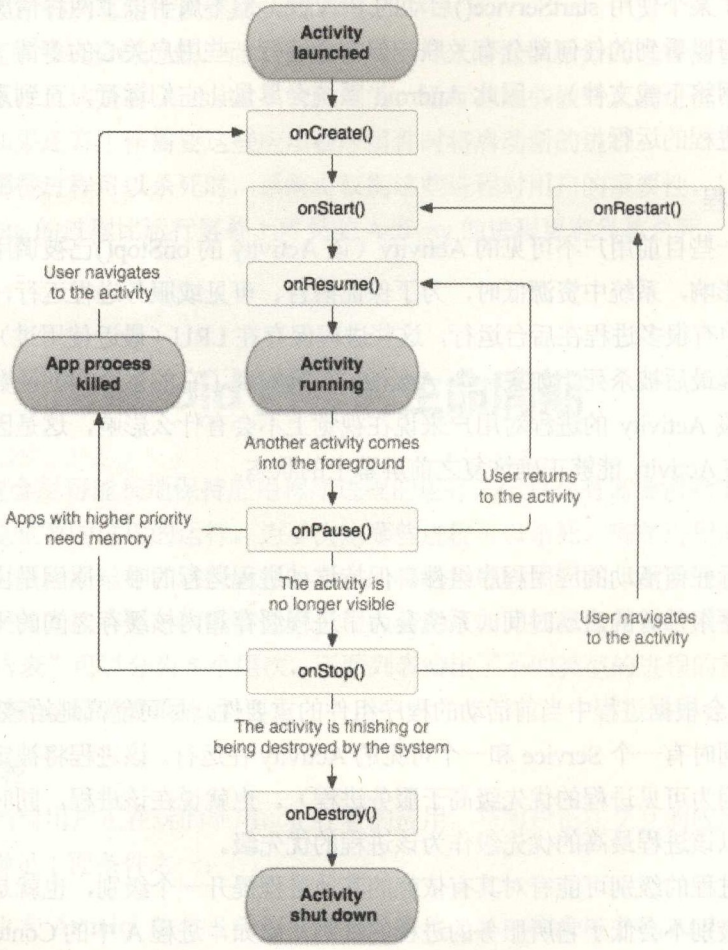
此外，一个进程的级别可能有对其有依赖的其他进程提升一个级别，也就是说给其他进程提供服务的进程的级别不会低于它所服务的进程的级别。比如，进程 A 中的 Content Provider 给进程 B 中某客户端提供数据服务，或者进程 A 中某个服务被进程 B 中某组件所绑定。那么进程 A 重要性程度不会低于进程 B。

由于运行 Service 的进程的级别高于运行后台 Activity 的进程的级别，一个需要较长时间运行操作的 Activity 启动能够完成该操作的 Service，可能也能很好地完成任务而无须简单创建一个新工作线程——尤其是该操作运行时间比该 Activity 还要长。比如，如果一个 Activity 需要完成向服务器上传图片任务时，应该使用一个服务来完成上载任务，这样即使用户离开该 Activity，Service 依然可以在后台完成上载任务。使用 Service 可以保证某个操作至少具有“服务进程”的优先级而

无须关心该 Activity 发生了什么变化。这也是一个 Broadcast Receiver 应该使用一个 Service 而非一线程来完成某个耗时的任务。

10.3 Android 中 Activity 的生命周期

Activity 是 Android 中必须要用的组件之一，我们来说一下它的生命周期，如下图所示。



activity 在系统中作为一种堆栈的方式进行管理。当一个新的活动开始，它被放在堆栈的顶部，成为正在运行的活动。之前运行中的 activity 都是在它下面的堆栈，当 top 上的 activity 退出的时候，堆栈里面它下面的 activity 会再次变成活动中的进程。

我们结合进程的生命周期和上面的图来说一下 activity 的实质的生命周期：

（1）当此应用变成“前台进程”的时候，对应的 activity 就会变成运行中的状态，依次调用 onCreate()、onStart()、onResume()方法。

（2）当此应用变成“可见进程”的时候，对应的 activity 就会变成暂停状态，就会调用 onPause()方法。

（3）当此应用变成“后台进程”的时候，对应的 activity 就会变成停止状态，就会相应地调用 onStop()方法。

（4）当此应用变成“销毁进程”的时候，对应的 activity 就会被回收，这时候就会调用 onDestroy()方法。

其实呢，activity 的生命周期的切换不仅发生在应用与应用之间，也发生在 activity 之间，由于不是本书的重点，我们就一笔带过了。下面对 activity 6 个方法分别说明：

（1）onCreate()方法——Activity 首次创建时最先被调用的方法，在 Activity 的一个完整生命周期中，此方法只会被调用一次。在开发过程中，我们一般需要使用 setContentView(int)方法来初始化 UI，对 UI 等进行数据绑定等操作。

（2）onStart()方法——onCreate()方法执行后被调用的方法，其一般和 onStop()一起组成 visible lifetime 的起始和终止阶段。在此阶段，用户对这个 Activity 是可见的但是不会获得焦点。

（3）onResume()方法——onStart()方法执行后被调用的方法，它和 onStop()方法一起组成 foreground lifetime。此方法执行完成后，Activity 可以获得用户的焦点，执行相应的方法。

（4）onPause()方法——当我们调用 startActivity(Intent)等方法启动另一个 Activity，且新 Activity 的 onCreate()方法调用之前会被调用的方法，当前 Activity 会调用此方法，用户对这个 Activity 将不可见。在此方法中，我们可以执行一些用来保存持久化的数据，停止动画，关闭一些耗时操作等的方法。这是启动一个新的 Activity 时一定会调用的方法。

（5）onStop()方法——此方法调用之后，我们将对这个 Activity 不再可见，所以如果新启动的 Activity 是一个全屏不透明的 Activity 时，这个方法将会被调用。

（6）onDestroy()——此方法一般是显示调用 finish()方法或者被系统强制销毁时，被调用的方法。这也是 Activity 生命周期的最后一个阶段。

10.4 Android 线程的运行机制

我觉得要理解 Android 的线程运行机制必须要理解两个线程：UI 主线程和工作线程。

1. UI 主线程

Android 系统启动某个应用后，将会创建一个线程来运行该应用，这个线程成为“主”线程。主线程非常重要，这是因为它要负责消息的分发，给界面上相应的 UI 组件分发事件，包括绘图事件。这也是应用可以和 UI 组件（为 android.widget 和 android.view 中定义的组件）发生直接交互的线程。因此主线程也通常称为用户界面线程（UI 主线程）。

Android 系统不会主动为应用程序的组件创建额外的线程。运用在同一进程中所有程序组件

都在 UI 线程中初始化，并使用 UI 线程来分发对这些程序组件的系统调用。由此可见，响应系统回调函数（比如 onKeyDown() 响应用户按键或者某个生命周期回调函数）的方法总是使用 UI 线程来运行。

比如，当用户触摸屏幕上某个按钮时，你的应用中的 UI 线程将把这个触摸事件发送到对应的 UI 小组件，然后该 UI 小组件设置其按下的状态并给事件队列发送一个刷新的请求，之后 UI 线程处理事件队列并通知该 UI 小组件重新绘制自身。

当你的应用中响应用户事件时需要完成一些费事的工作时，这种单线程工作模式可能会导致非常差的用户响应性能。尤其是如果所有的工作都在 UI 线程中完成，比如访问网络，数据库查询等费时的工作将会阻塞 UI 线程。当 UI 线程被阻塞时，就无法分发事件，包括绘图事件。此时从用户的角度来看，该应用看起来不再有响应。更为糟糕的是，如果 UI 线程阻塞超过几秒钟（目前为五秒），系统将给用户显示著名的“应用程序无响应（ANR）”对话框。用户可能会选择退出你的应用，更为甚者如果他们感觉很不满意也会选择卸载你的应用。

此外，Android 的 UI 组件包不是“线程安全”的，因此你不能走工作线程中调用 UI 组件的方法，所有有关 UI 的操作必须在 UI 线程中完成，因此，下面给出使用 UI 单线程工作线程的两个规则：

- 永远不要阻塞 UI 线程。
- 不要在非 UI 线程中操作 UI 组件。

2. 工作线程

由于 Android 使用单线程工作模式，因此不阻塞 UI 线程对于应用程序的响应性能至关重要。如果在你的应用中包含一些不是一瞬间就能完成的操作的话，你应该使用额外的线程（工作线程或是后台线程）来执行这些操作。

比如下面示例，在用户单击某个按钮后，就启动一个新线程来下载某个图像，然后在 ImageView 中显示：

```
public void onClick(View v) {
        new Thread(new Runnable() {
            public void run() {
                                                Bitmap                    b
= loadImageFromNetwork("http://example.com/image.png");
                mImageView.setImageBitmap(b);
            }
        }).start();
}
```

初看，这段代码应该很好地完成工作，因为它创建了一个新线程来完成网络操作。然而它违反了上面说的第二个规则：不要在非 UI 线程中操作 UI 组件。在这段代码中的工作线程中而不是在 UI 线程中，直接修改 ImageView，这将导致一些不可以预见的后果，常常导致发现此类错误捕

捉异常困难和费时。

为此解决上面的问题和错误，Android 提供了多种方法使得在非 UI 线程中访问 UI 组件，下面给出了其中的几种方法：

- Activity.runOnUiThread(Runnable)
- View.post(Runnable)
- View.postDelayed(Runnable, long)

比如，使用 View.post(Runnable)修改上面的代码如下：

```
public void onClick(View v) {
        new Thread(new Runnable() {
            public void run() {
                final Bitmap                                                    bitmap
= loadImageFromNetwork("http://example.com/image.png");
                mImageView.post(new Runnable() {
                    public void run() {
                        mImageView.setImageBitmap(bitmap);
                    }
                });
            }
        }).start();
}
```

这样的实现是符合“线程安全”原则的：在额外的线程中完成网络操作，并且在 UI 线程中完成对 ImageView 的操作。

然而，随着操作复杂性的增加，上述代码可能会变得非常复杂而导致维护困难。为了解决工作线程中处理此类复杂操作，你可能会考虑在工作线程中使用 Handler 类来处理由 UI 线程发送过来的消息。但是，可能使用 AsyncTask 是此类问题的最好解决方案，它很好地简化了工作线程需要和 UI 组件发生交互的问题。

10.5 Android 异步线程的处理方法

AsyncTask 允许你完成一些和用户界面相关的异步工作。它在一个工作线程中完成一些阻塞工作任务，然后在任务完成后通知 UI 线程，这些都不需要你自己来管理工作线程。

你必须从 AsyncTask 派生一个子类并实现 doInBackground()回调函数来使用 AsyncTask，AsyncTask 将使用后台进程池来执行异步任务。为了能够更新用户界面，你必须实现 onPostExecute()方法，该方法将传递 doInBackground()的返回结果，并且运行在 UI 线程中。然后你可以在 UI 线

程中调用 execute() 方法来执行该任务。

比如，使用 AsyncTask 来完成之前的例子：

```
public void onClick(View v) {
        new DownloadImageTask().execute("http://example.com/image.png");
}
private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
        //传递参数给 worker thread,来源于 AsyncTask.execute()
        protected Bitmap doInBackground(String... urls) {
            return loadImageFromNetwork(urls[0]);
        }
        //AsyncTask 完成任务后回调的方法
        protected void onPostExecute(Bitmap result) {
            mImageView.setImageBitmap(result);
        }
}
```

现在 UI 是安全的而且代码变得更简单，因为它把在工作线程中的工作和在 UI 线程的工作很好地分隔开来了。

由于本书的重点不在于此，就简单地说一下 AsyncTask 的常用方法：

（1）你可以使用 generics 为 Task 指定参数类型，返回值类型等。

（2）方法 doInBackground()将自动在一个工作线程中执行。

（3）方法 onPreExecute()、onPostExecute()和 onProgressUpdate 都在 UI 线程中调用。

（4）方法 doInBackground()的返回值将传递给 onPostExecute()方法。

（5）你可以在 doInBackground()中任意调用 publishProgress()方法，该方法将会调用 UI 线程中的 onProgressUpdate()方法，你可以用它来报告任务完成的进度。

（6）你可以在任意线程中任意时刻终止任务的执行。

（7）要注意的是，由于系统配置的变化（比如屏幕的方向转动），你的工作线程可能会碰到意外的重新启动，这种情况下，你的工作线程可能被销毁，你可以参考 Android 开发包中 Shelves 示例来处理线程重新的问题。

10.6 Android 异步线程的原理与实现

我们结合前几章学过的线程池的原理来分析一下 AsyncTask 的源码。

```
public abstract class AsyncTask<Params, Progress, Result> {
```

（1） AsyncTask 里面的构造方法利用了我们前面 future 的线程阀机制，创建了一个异步的工作线程。

```
public AsyncTask() {
        //构造方法利用我们的前面讲到的 Callable 和 FutureTask 机制，实现由返回结果的异步线程执行。
        mWorker = new WorkerRunnable<Params, Result>() {
            public Result call() throws Exception {
                mTaskInvoked.set(true );
                Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
                //有返回结果的线程执行，调用 doInBackground 方法内容。
                return postResult(doInBackground(mParams ));
            }
        };
        //get 的时候如果没有得到结果，下面不会执行的线程阀机制
        mFuture = new FutureTask<Result>(mWorker) {
            @Override
            protected void done() {
                try {
                    //发送一个做完的消息通知
                    postResultIfNotInvoked(get());
                } catch (InterruptedException e) {
                    android.util.Log. w(LOG_TAG , e);
                } catch (ExecutionException e) {
                    throw new RuntimeException("An error occured while executing doInBackground()",e.getCause());
                } catch (CancellationException e) {
                    postResultIfNotInvoked( null);
                }
            }
        };
    }
```

（2）查看 execute 方法。

```
public final AsyncTask<Params, Progress, Result> execute(Params... params) {
        return executeOnExecutor(sDefaultExecutor, params);
    }
    public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec,
```

```
            Params... params) {
        if (mStatus != Status.PENDING) {
            switch (mStatus ) {
                case RUNNING :
                    throw new IllegalStateException("Cannot execute task:"
                            + " the task is already running." );
                case FINISHED :
                    throw new IllegalStateException("Cannot execute task:"
                            + " the task has already been executed "
                            + "(a task can be executed only once)" );
            }
        }
        mStatus = Status.RUNNING;
        onPreExecute(); //可以重写此方法在执行之前执行
        //这里给我们构造函数里面创建的 mWorker 线程传参数
        mWorker.mParams = params;
        exec.execute( mFuture);//利用线程池
        return this ;
    }
```

（3）查看一下线程池是如何创建和使用的。

```
    //CPU 数量
    private static final int CPU_COUNT =
Runtime.getRuntime().availableProcessors();
    //线程池的核心大小 CPU 数量+1
    private static final int CORE_POOL_SIZE = CPU_COUNT + 1;
    //线程池的最大值 CPU 数量*2 +1
    private static final int MAXIMUM_POOL_SIZE = CPU_COUNT * 2 + 1;
    private static final int KEEP_ALIVE = 1;
    //线程工厂类，方便线程池创建线程使用
    private static final ThreadFactory sThreadFactory = new ThreadFactory() {
        private final AtomicInteger mCount = new AtomicInteger(1);
        public Thread newThread(Runnable r) {
            return new Thread(r, "AsyncTask #" + mCount.getAndIncrement());
        }
    };
    //采用 LinkedBlockingQueue 做线程队列
    private static final BlockingQueue<Runnable> sPoolWorkQueue =
            new LinkedBlockingQueue<Runnable>(128);
```

```
    //类加载完初始化一个线程池
    public static final Executor THREAD_POOL_EXECUTOR
            = new ThreadPoolExecutor(CORE_POOL_SIZE, MAXIMUM_POOL_SIZE, KEEP_AL
IVE,
                    TimeUnit. SECONDS, sPoolWorkQueue , sThreadFactory);
    //Executor 线程池的 Util 类
    public static final Executor SERIAL_EXECUTOR = new SerialExecutor();
    private static volatile Executor sDefaultExecutor = SERIAL_EXECUTOR;

    private static class SerialExecutor implements Executor {
        final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
        Runnable mActive;
        public synchronized void execute(final Runnable r) {
            mTasks.offer(new Runnable() {
                public void run() {
                    try {
                        r.run();
                    } finally {
                        scheduleNext();
                    }
                }
            });
            if (mActive == null) {
                scheduleNext();
            }
        }
        protected synchronized void scheduleNext() {
            if ((mActive = mTasks.poll()) != null) {
                //引用初始化加载的线程池
                THREAD_POOL_EXECUTOR.execute( mActive);
            }
        }
    }
```

（4）我们来看一下 onPostExecute 是什么时间被调用的。

我们在前面看到 FutureTask 的 done 方法在 thread 完成之后会调用 postResultIfNotInvoked 方法。

```
    private void postResultIfNotInvoked(Result result) {
        final boolean wasTaskInvoked = mTaskInvoked.get();
```

```
        if (!wasTaskInvoked) {
            postResult(result);
        }
    }
    private Result postResult(Result result) {
        //交由 sHandler 来管理 UI 主线的 loop 机制
        Message message = sHandler.obtainMessage(MESSAGE_POST_RESULT,
                new AsyncTaskResult<Result>(this, result) );
        message.sendToTarget();
        return result;
    }
    private static class InternalHandler extends Handler {
        @SuppressWarnings({"unchecked" , "RawUseOfParameterizedType"})
        @Override
        public void handleMessage(Message msg) {
            AsyncTaskResult result = (AsyncTaskResult) msg.obj ;
            switch (msg.what ) {
                case MESSAGE_POST_RESULT :
                    // 执行 AsyncTask 任务的 finish 方法,
                    result. mTask.finish(result.mData [0]);
                    break;
                case MESSAGE_POST_PROGRESS :
                    result. mTask.onProgressUpdate(result.mData );
                    break;
            }
        }
    }
    private void finish(Result result) {
        if (isCancelled()) {
            onCancelled(result);
        } else {
            //如果执行成功，没有被取消执行 onPostExecute 方法更新 UI 主线程
            onPostExecute(result);
        }
        mStatus = Status.FINISHED;
    }
……//关于 sHandler 和消息机制我们这里就不多说了。
}
```

通过第三步，我们可以看到线程池的 core 大小一般默认都是 CPU 的数量加 1，最大线程池数量是 CPU 数量的 2 倍加 1。其实线程、线程池、并发编程无处不在。最后，如果大家能了解 Android 的运行机理，基本上由 Java 工程师转 Android 就非常简单了，加上一些事件处理逻辑和 UI 的相关显示，基本上就可以完成 Android 开发了。

附录1
JVM的参数

参数名称	含义	默认值	
-Xms	初始堆大小	物理内存的1/64(<1GB)	默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制
-Xmx	最大堆大小	物理内存的1/4(<1GB)	默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制
-Xmn	年轻代大小(1.4or lator)		注意：此处的大小是（eden+ 2 survivor space)，与jmap -heap中显示的New gen是不同的。 整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。 增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8
-XX:NewSize	设置年轻代大小(for 1.3/1.4)		
-XX:MaxNewSize	年轻代最大值(for 1.3/1.4)		
-XX:PermSize	设置持久代(perm gen)初始值	物理内存的1/64	
-XX:MaxPermSize	设置持久代最大值	物理内存的1/4	
-Xss	每个线程的堆栈大小		JDK5.0以后每个线程堆栈大小为1MB，以前每个线程堆栈大小为256KB。更具应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右 一般小的应用，如果栈不是很深， 应该是128KB够用的，大的应用建议使用256KB。这个选项对性能影响比较大，需要严格地测试。 和threadstacksize选项解释很类似，官方文档似乎没有解释，在论坛中有这样一句话：“-Xss is translated in a VM flag named ThreadStackSize”一般设置这个值就可以了

（续表）

参数名称	含义	默认值	
-XX:ThreadStackSize	Thread Stack Size		(0 means use default stack size) [Sparc: 512; Solaris x86: 320 (was 256 prior in 5.0 and earlier); Sparc 64 bit: 1024; Linux amd64: 1024 (was 0 in 5.0 and earlier); all others 0.]
-XX:NewRatio	年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）		-XX:NewRatio=4 表示年轻代与年老代所占比值为 1:4，年轻代占整个堆栈的 1/5 Xms=Xmx 并且设置了 Xmn 的情况下，该参数不需要进行设置
-XX:SurvivorRatio	Eden 区与 Survivor 区的大小比值		设置为 8，则两个 Survivor 区与一个 Eden 区的比值为 2:8，一个 Survivor 区占整个年轻代的 1/10
-XX:LargePageSizeInBytes	内存页的大小不可设置过大， 会影响 Perm 的大小		=128m
-XX:+UseFastAccessorMethods	原始类型的快速优化		
-XX:+DisableExplicitGC	关闭 System.gc()		这个参数需要严格地测试
-XX:MaxTenuringThreshold	垃圾最大年龄		如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象在年轻代的存活时间，增加在年轻代即被回收的概率该参数只有在串行 GC 时才有效
-XX:+AggressiveOpts	加快编译		
-XX:+UseBiasedLocking	锁机制的性能改善		
-Xnoclassgc	禁用垃圾回收		
-XX:SoftRefLRUPolicyMSPerMB	每兆堆空闲空间中 SoftReference 的存活时间	1s	softly reachable objects will remain alive for some amount of time after the last time they were referenced. The default value is one second of lifetime per free megabyte in the heap
-XX:PretenureSizeThreshold	对象超过多大是直接在旧生代分配	0	单位字节 新生代采用 Parallel Scavenge GC 时无效 另一种直接在旧生代分配的情况是大的数组对象，且数组中无外部引用对象
-XX:TLABWasteTargetPercent	TLAB 占 eden 区的百分比	1%	
-XX:+*CollectGen0First*	FullGC 时是否先 YGC	false	

并行收集器相关参数

参数名称	含义	默认值	
-XX:+UseParallelGC	Full GC 采用 parallel MSC（此项待验证）		选择垃圾收集器为并行收集器。此配置仅对年轻代有效，即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集
-XX:+UseParNewGC	设置年轻代为并行收集		可与 CMS 收集同时使用 JDK5.0 以上，JVM 会根据系统配置自行设置，所以无须再设置此值
-XX:ParallelGCThreads	并行收集器的线程数		此值最好配置与处理器数目相等 同样适用于 CMS
-XX:+UseParallelOldGC	年老代垃圾收集方式为并行收集（Parallel Compacting）		这个是 JAVA 6 出现的参数选项
-XX:MaxGCPauseMillis	每次年轻代垃圾回收的最长时间（最大暂停时间）		如果无法满足此时间，JVM 会自动调整年轻代大小，以满足此值
-XX:+UseAdaptiveSizePolicy	自动选择年轻代区大小和相应的 Survivor 区比例		设置此选项后，并行收集器会自动选择年轻代区大小和相应的 Survivor 区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开
-XX:GCTimeRatio	设置垃圾回收时间占程序运行时间的百分比		公式为 1/(1+n)
-XX:+*ScavengeBeforeFullGC*	Full GC 前调用 YGC	true	Do young generation GC prior to a full GC. (Introduced in 1.4.1.)

CMS 相关参数

参数名称	含义	默认值	
-XX:+UseConcMarkSweepGC	使用 CMS 内存收集		测试中配置这个以后，-XX:NewRatio=4 的配置失效了，原因不明。所以，此时最好使用-Xmn 设置
-XX:+AggressiveHeap			试图是使用大量的物理内存 长时间大内存使用的优化，能检查计算资源（内存， 处理器数量） 至少需要 256MB 内存 大量的 CPU / 内存，（在 1.4.1 在 4CPU 的机器上已经显示有提升）

（续表）

参数名称	含义	默认值	
-XX:CMSFullGCsBeforeCompaction	多少次后进行内存压缩		由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 GC 以后对内存空间进行压缩，整理
-XX:+CMSParallelRemarkEnabled	降低标记停顿		
-XX+UseCMSCompactAtFullCollection	在 FULL GC 的时候对年老代的压缩		CMS 是不会移动内存的，因此，这个非常容易产生碎片，导致内存不够用，因此，内存的压缩这个时候就会被启用。增加这个参数是个好习惯。 可能会影响性能，但是可以消除碎片
-XX:+UseCMSInitiatingOccupancyOnly	使用手动定义初始化定义开始 CMS 收集		禁止 hostspot 自行触发 CMS GC
-XX:CMSInitiatingOccupancyFraction=70	使用 cms 作为垃圾回收 使用 70％后开始 CMS 收集	92	为了保证不出现 promotion failed(见下面介绍)错误，该值的设置需要满足以下计算公式 CMSInitiatingOccupancyFraction
-XX:CMSInitiatingPermOccupancyFraction	设置 Perm Gen 使用到达多少比率时触发	92	
-XX:+CMSIncrementalMode	设置为增量模式		用于单 CPU 情况
-XX:+HeapDumpOnOutOfMemoryError	当放生 out of MemoryError 的时候自动生成 dump 文件		默认关闭

辅助信息

参数名称	含义	默认值	
-XX:+PrintGC			输出形式: [GC 118250K->113543K(130112K), 0.0094143 secs] [Full GC 121376K->10414K(130112K), 0.0650971 secs]
-XX:+PrintGCDetails			输出形式:[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs] [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs]

（续表）

参数名称	含义	默认值	
-XX:+PrintGCTimeStamps			
-XX:+PrintGC:PrintGCTimeStamps			可与-XX:+PrintGC -XX:+PrintGCDetails 混合使用 输出形式:11.851: [GC 98328K->93620K(130112K), 0.0082960 secs]
-XX:+PrintGCApplicationStoppedTime	打印垃圾回收期间程序暂停的时间。可与上面混合使用		输出形式:Total time for which application threads were stopped: 0.0468229 seconds
-XX:+PrintGCApplicationConcurrentTime	打印每次垃圾回收前，程序未中断的执行时间。可与上面混合使用		输出形式:Application time: 0.5291524 seconds
-XX:+PrintHeapAtGC	打印 GC 前后的详细堆栈信息		
-Xloggc:filename	把相关日志信息记录到文件以便分析。与上面几个配合使用		
-XX:+PrintClassHistogram	garbage collects before printing the histogram.		
-XX:+PrintTLAB	查看 TLAB 空间的使用情况		
XX:+PrintTenuringDistribution	查看每次 minor GC 后新的存活周期的阈值		Desired survivor size 1048576 bytes, new threshold 7 (max 15) new threshold 7 即标识新的存活周期的阈值为 7

附录2
jstat的语法

- jstat -class pid:显示加载 class 的数量，及所占空间等信息。
- jstat -compiler pid:显示 VM 实时编译的数量等信息。
- jstat -gc pid:可以显示 gc 的信息，查看 gc 的次数，及时间。其中最后五项，分别是 young gc 的次数，young gc 的时间，full gc 的次数，full gc 的时间，gc 的总时间。
- jstat -gccapacity:可以显示，VM 内存中三代（young,old,perm）对象的使用和占用大小，如：PGCMN 显示的是最小 perm 的内存使用量，PGCMX 显示的是 perm 的内存最大使用量，PGC 是当前新生成的 perm 内存占用量，PC 是当前 perm 内存占用量。其他的可以根据这个类推， OC 是 old 内存的占用 量。
- jstat -gcnew pid: new 对象的信息。
- jstat -gcnewcapacity pid: new 对象的信息及其占用量。
- jstat -gcold pid: old 对象的信息。
- jstat -gcoldcapacity pid: old 对象的信息及其占用量。
- jstat -gcpermcapacity pid: perm 对象的信息及其占用量。
- jstat -util pid: gc 信息统计。
- jstat -printcompilation pid: 当前 VM 执行的信息。

除了以上一个参数外，还可以同时加上两个数字，如：jstat -printcompilation 3024 250 6 是每250ms 打印一次，一共打印 6 次，还可以加上-h3 每三行显示一下标题。

语法结构：

```
Usage: jstat -help|-options
jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]
```

参数解释：

- Options——选项，我们一般使用 -gcutil 查看 gc 情况。
- vmid——VM 的进程号，即当前运行的 java 进程号。
- interval——间隔时间，单位为秒或者 ms。
- count——打印次数，如果默认则打印无数次。

实例如下：

```
#sudo /app/jdk1.7.0_45/bin/jstat -gcutil -h 10 4242 1000
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
```

```
100.00    0.00 100.00 100.00  59.75    207    6.948    55    67.452    74.400
100.00    0.00 100.00 100.00  59.75    207    6.948    55    67.452    74.400
100.00    0.00 100.00 100.00  59.75    207    6.948    55    67.452    74.400
100.00    0.00 100.00 100.00  59.75    207    6.948    55    67.452    74.400
100.00    0.00 100.00 100.00  59.75    207    6.948    55    67.452    74.400
100.00    0.00 100.00 100.00  59.75    207    6.948    55    67.452    74.400
100.00    0.00 100.00 100.00  59.75    207    6.948    55    67.452    74.400
100.00    0.00 100.00 100.00  59.75    207    6.948    55    67.452    74.400
```

附录3
jstat中一些术语的中文解释

S0C：年轻代中第一个 survivor（幸存区）的容量（字节）。
S1C：年轻代中第二个 survivor（幸存区）的容量（字节）。
S0U：年轻代中第一个 survivor（幸存区）目前已使用空间（字节）。
S1U：年轻代中第二个 survivor（幸存区）目前已使用空间（字节）。
EC：年轻代中 Eden（伊甸园）的容量（字节）。
EU：年轻代中 Eden（伊甸园）目前已使用空间（字节）。
OC：年老代的容量（字节）。
OU：年老代目前已使用空间（字节）。
PC：Perm（持久代）的容量（字节）。
PU：Perm（持久代）目前已使用空间（字节）。
YGC：从应用程序启动到采样时年轻代中 gc 次数。
YGCT：从应用程序启动到采样时年轻代中 gc 所用时间（s）。
FGC：从应用程序启动到采样时年老代（全 gc）gc 次数。
FGCT：从应用程序启动到采样时年老代（全 gc）gc 所用时间（s）。
GCT：从应用程序启动到采样时 gc 用的总时间（s）。
NGCMN：年轻代（young）中初始化（最小）的大小（字节）。
NGCMX：年轻代（young）的最大容量（字节）。
NGC：年轻代（young）中当前的容量（字节）。
OGCMN：年老代中初始化（最小）的大小（字节）。
OGCMX：年老代的最大容量（字节）。
OGC：年老代当前新生成的容量（字节）。
PGCMN：perm 代中初始化（最小）的大小（字节）。
PGCMX：perm 代的最大容量（字节）。
PGC：perm 代当前新生成的容量（字节）。
S0：年轻代中第一个 survivor（幸存区）已使用的占当前容量百分比。
S1：年轻代中第二个 survivor（幸存区）已使用的占当前容量百分比。
E：年轻代中 Eden（伊甸园）已使用的占当前容量百分比。
O：old 代已使用的占当前容量百分比。
P：perm 代已使用的占当前容量百分比。
S0CMX：年轻代中第一个 survivor（幸存区）的最大容量（字节）。

S1CMX ：年轻代中第二个 survivor（幸存区）的最大容量（字节）。

ECMX：年轻代中 Eden（伊甸园）的最大容量（字节）。

DSS：当前需要 survivor（幸存区）的容量（字节）（Eden 区已满）。

TT：持有次数限制。

MTT：最大持有次数限制。

附录4

Tomcat配置文件server.xml中Executor的参数

1. /conf/server.xml 中 Executor 对应的相关参数

```
<Executor name="tomcatThreadPool" namePrefix="catalina-exec-"
maxThreads="500" minSpareThreads="20" maxIdleTime="60000" />
```

参数	说明
lassName	线程池的实现类，如果自定义要实现 org.apache.catalina.Executor 接口。不填的情况下，默认 org.apache.catalina.core.StandardThreadExecutor
name	线程池的名字，当然 server.xml 可以定义多个线程池
threadPriority	线程的优先级，默认 5
daemon	是否是守护线程，默认 true
namePrefix	线程的名字前缀，线程的名字有 namePrefix+threadNumber 组成。我们上面设置的是“catalina-exec-”
maxThreads	线程池最大的线程数量，默认 200。类似线程池里面的 corePoolSize
minSpareThreads	线程池的永远活动的线程的数量，默认 25。类似线程池里面的 maximumPoolSize
maxIdleTime	minSpareThreads 到 maxThreads 之前的线程最长存活时间，默认 60000ms
maxQueueSize	线程队列的最大值。我们前面不是说过 queue 吗，超过这个队列再进来的线程就会直接抛弃处理，默认：Integer.MAX_VALUE

2. /conf/server.xml 中 Connector 对应的相关参数

```
<Connector executor="tomcatThreadPool"
               port="80" protocol="HTTP/1.1"
               connectionTimeout="60000"
               keepAliveTimeout="15000"
               maxKeepAliveRequests="1"
               redirectPort="443"
                        maxHttpHeaderSize="8192"          URIEncoding="UTF-8"
enableLookups="false" acceptCount="100" disableUploadTimeout="true"/>
```

参数	说明
allowTrace	是否运行跟踪 http 的方法，默认 false;
connectionTimeout	网络连接超时，单位：ms。设置为 0 表示永不超时，这样设置有隐患的。通常可设置为 30000ms
keepAliveTimeout	长连接最大保持时间（ms）。此处为 15s
maxKeepAliveRequests	最大长连接个数(1 表示禁用，-1 表示不限制个数，默认 100 个。一般设置在 100~200 之间)
maxHttpHeaderSize	http 请求头信息的最大程度，超过此长度的部分不予处理。一般 8KB
URIEncoding	指定 Tomcat 容器的 URL 编码格式
acceptCount	指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理，默认为 10 个
disableUploadTimeout	上传时是否使用超时机制
port	端口
protocol	协议 org.apache.coyote.http11.Http11Protocol 和 HTTP/1.1 相同（默认值） org.apache.coyote.http11.Http11NioProtocol org.apache.coyote.http11.Http11AprProtocol
enableLookups	是否反查域名，取值为：true 或 false。为了提高处理能力，应设置为 false
bufferSize	默认 2048 bytes
maxSpareThreads	做多空闲连接数，一旦创建的线程超过这个值，Tomcat 就会关闭不再需要的 socket 线程。默认 50
maxThreads	最多同时处理的连接数，Tomcat 使用线程来处理接收的每个请求。这个值表示 Tomcat 可创建的最大的线程数。默认 200
minSpareThreads	最小空闲线程数，Tomcat 初始化时创建的线程数。默认 4

附录5
Thread的API

嵌套类摘要	
static class Thread.State	线程状态
static interface Thread.UncaughtExceptionHandler	当 Thread 因未捕获的异常而突然终止时，调用处理程序的接口

字段摘要	
static int MAX_PRIORITY	线程可以具有的最高优先级
static int MIN_PRIORITY	线程可以具有的最低优先级
static int NORM_PRIORITY	分配给线程的默认优先级

方法摘要	
Thread()	分配新的 Thread 对象
Thread(Runnable target)	分配新的 Thread 对象
Thread(Runnable target, String name)	分配新的 Thread 对象
Thread(String name)	分配新的 Thread 对象
Thread(ThreadGroup group, Runnable target)	分配新的 Thread 对象
Thread(ThreadGroup group, Runnable target, String name)	分配新的 Thread 对象，以便将 target 作为其运行对象，将指定的 name 作为其名称，并作为 group 所引用的线程组的一员
Thread(ThreadGroup group, Runnable target, String name, long stackSize)	分配新的 Thread 对象，以便将 target 作为其运行对象，将指定的 name 作为其名称，作为 group 所引用的线程组的一员，并具有指定的堆栈大小
Thread(ThreadGroup group, String name)	分配新的 Thread 对象

方法摘要	
static int activeCount()	返回当前线程的线程组中活动线程的数目
Void checkAccess()	判定当前运行的线程是否有权修改该线程
Int countStackFrames()	不建议使用。该调用的定义依赖于 suspend()，但它遭到了反对。此外，该调用的结果从来都不是意义明确的
static Thread currentThread()	返回对当前正在执行的线程对象的引用

（续表）

方法摘要	
Void destroy()	已过时。该方法最初用于破坏该线程，但不作任何清除。它所保持的任何监视器都会保持锁定状态。不过，该方法决不会被实现。即使要实现，它也极有可能以 suspend() 方式被死锁。如果目标线程被破坏时保持一个保护关键系统资源的锁，则任何线程在任何时候都无法再次访问该资源。如果另一个线程曾试图锁定该资源，则会出现死锁。这类死锁通常会证明它们自己是“冻结”的进程。有关更多信息，请参阅为何不赞成使用 Thread.stop、Thread.suspend 和 Thread.resume?
static void dumpStack()	将当前线程的堆栈跟踪打印至标准错误流
static int enumerate(Thread[] tarray)	将当前线程的线程组及其子组中的每一个活动线程复制到指定的数组中
static Map<Thread,StackTraceElement[]> getAllStackTraces()	返回所有活动线程的堆栈跟踪的一个映射
ClassLoader getContextClassLoader()	返回该线程的上下文 ClassLoader
static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()	返回线程由于未捕获到异常而突然终止时调用的默认处理程序
Long getId()	返回该线程的标识符
String getName()	返回该线程的名称
Int getPriority()	返回线程的优先级
StackTraceElement[]getStackTrace()	返回一个表示该线程堆栈转储的堆栈跟踪元素数组
Thread.State getState()	返回该线程的状态
ThreadGroup getThreadGroup()	返回该线程所属的线程组
Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()	返回该线程由于未捕获到异常而突然终止时调用的处理程序
static boolean holdsLock(Object obj)	当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true
Void interrupt()	中断线程
static boolean interrupted()	测试当前线程是否已经中断
Boolean isAlive()	测试线程是否处于活动状态
Boolean isDaemon()	测试该线程是否为守护线程
Boolean isInterrupted()	测试线程是否已经中断
Void join()	等待该线程终止
Void join(long millis)	等待该线程终止的时间最长为 millis ms
Void join(long millis, int nanos)	等待该线程终止的时间最长为 millis ms + nanos ns
Void resume()	不建议使用。该方法只与 suspend() 一起使用，但 suspend() 已经遭到反对，因为它具有死锁倾向
Void run()	如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回

（续表）

方法摘要	
Void setContextClassLoader(ClassLoader cl)	设置该线程的上下文 ClassLoader
Void setDaemon(boolean on)	将该线程标记为守护线程或用户线程
static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)	设置当线程由于未捕获到异常而突然终止，并且没有为该线程定义其他处理程序时所调用的默认处理程序
Void setName(String name)	改变线程名称，使之与参数 name 相同
Void setPriority(int newPriority)	更改线程的优先级
Void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)	设置该线程由于未捕获到异常而突然终止时调用的处理程序
static void sleep(long millis)	在指定的 ms 数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响
static void sleep(long millis, int nanos)	在指定的 ms 数加指定的纳秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响
Void start()	使该线程开始执行；Java 虚拟机调用该线程的 run 方法
Void stop()	不建议使用。该方法具有固有的不安全性。用 Thread.stop 来终止线程将释放它已经锁定的所有监视器（作为沿堆栈向上传播的未检查 ThreadDeath 异常的一个自然后果）。如果以前受这些监视器保护的任何对象都处于一种不一致的状态，则损坏的对象将对其他线程可见，这有可能导致任意的行为。stop 的许多使用都应由只修改某些变量以指示目标线程应该停止运行的代码来取代。目标线程应定期检查该变量，并且如果该变量指示它要停止运行，则从其运行方法依次返回。如果目标线程等待很长时间（例如基于一个条件变量），则应使用 interrupt 方法来中断该等待
Void stop(Throwable obj)	不建议使用。该方法具有固有的不安全性
Void suspend()	不建议使用。该方法已经遭到反对，因为它具有固有的死锁倾向。如果目标线程挂起时在保护关键系统资源的监视器上保持有锁，则在目标线程重新开始以前任何线程都不能访问该资源。如果重新开始目标线程的线程想在调用 resume 之前锁定该监视器，则会发生死锁。这类死锁通常会证明自己是“冻结”的进程
String toString()	返回该线程的字符串表示形式，包括线程名称、优先级和线程组
static void yield()	暂停当前正在执行的线程对象，并执行其他线程

结束语

多监控，多结合实际情况查看，相信不出多久就会理解的很深刻。任何语言都必须要掌握起源和根基，感谢大家的支持，有任何问题欢迎加入 QQ 群交流。

优秀的人在找煎熬，普通的人在找舒服。最后，祝愿各位读者都能高天阔地展雄才，举步生辉创伟业！

相关推荐的书籍有：

《深入理解 JVM 虚拟机》

《大数据时代》

《深入剖析 Tomcat》

《我也能做 CTO 之程序员职业规划》

《TCP/IP Socket in Java》

Java 高端交流 QQ 群：240619787，欢迎大家一起交流和学习。

书中源码的下载地址：【http://pan.baidu.com/s/1mgzIbX2】

作者：张振华.Jack

二维码地址如下：

Java并发编程从入门到精通

对于一个初学者，或者是工作了几年的Java工程师来说，通过详细地研读此书，相信一定会有或多或少的收获。看到Jack几个月来辛勤的劳动成果，感到钦佩。本书几乎涵盖了所有Java多并发、多线程开发相关的学习资料。相信此书一定会成为一个很好的多并发方面的、书不离手的开发手册。

Alibaba高级开发工程师　Steve.Xiu

韩愈有言“师者，所以传道受业解惑也”，读完此书，对作者肃然起敬，他不正是秉承了这样的师道在作此书嘛！十多年的开发经验倾囊相授，言之谆谆，例之凿凿，实为难得。对Java多线程的知识讲解得如此详尽，如此深入，给人一种一览众山小的感觉。期待更多力作面世！

知名互联网公司项目经理 XiaoShuang.Li（李小双）

清华大学出版社数字出版网站
WQBook 书文局泉
www.wqbook.com

ISBN 978-7-302-40191-9
9 787302 401919 >
定价：39.00元